VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Framework pro vykreslování grafů pro platformu iOS

Graph Framework for iOS Platform

2012 Jiří Urbášek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jiří Urbášek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Framework pro vykreslování grafů pro platformu iOS
Graph Framework for iOS Platform

Zásady pro vypracování:

Cílem práce je vytvořit framework pro zobrazování a obsluhu grafů pro platformu iOS, který bude samostatný a bude jej možné použít v různých aplikacích pro iPhone či iPad.

1. Porovnejte existující grafové frameworky, uveďte jejich výhody a nevýhody.
2. Popiště možnosti vykreslování grafů na iOS zařízeních (iPhone a iPad).
3. Navrhněte strukturu a rozhraní grafového frameworku.
4. Popište různé typy grafů a prozkoumejte jejich možnosti interakce s uživatelem (animace, dotyky, gesta atp.).
5. Ukažte možnosti interakce grafu s okolním prostředím (při změně grafu se změní data v přidružené tabulce atp.).
6. Prozkoumejte možnosti exportu grafů (např. do .png nebo .pdf pro odeslání emailem atp.).

Seznam doporučené odborné literatury:

Mark, David, Beginning iPhone 4 Development: Exploring the iOS SDK, Apress; 1st edition, 2011, ISBN 1430230243 Neuburg, Matt, Programming iOS 4: Fundamentals of iPhone, iPad, and iPod Touch Development, O'Reilly Media, 2011, ISBN 1449388434 Lewis, Rory, iPhone and iPad Apps for Absolute Beginners, Apress, 2009, ISBN 1430227001

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Mgr. Michal Šrajer**

Konzultant diplomové práce: Ing. Michal Krumnikl

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 4. května 2012 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2012 .....

Na tomto místě bych rád poděkoval Mgr. Michalovi Šrajerovi za cenné rady a připomínky během vedení mé diplomové práce. Ing. Michalovi Krumniklovi za doporučení a rady při psaní tohoto textu. Ing. et Ing. Milanovi Janáskovi CSc. a Květoslavě Liškové za pomoc při zdokonalování mého jazykového projevu. A v neposlední řadě také mým rodičům, kteří mi poskytli láskyplné zázemí a umožnili mi tak tuto práci zdárně dokončit.

Abstrakt

Platforma iOS je v dnešní době jedním z nevýznamnějších hráčů na poli operačních systémů mobilních zařízení. iOS aplikace se začínají využívat i ke zpracovávání a vizualizaci dat. Pro přehledné zobrazení dat se velmi často používají grafy, avšak pro platformu iOS prozatím neexistuje žádná volně dostupná grafová knihovna, která by poskytovala uspokojivé výstupy. Z tohoto důvodu vznikla tato diplomová práce, která se snaží zmapovat situaci v oblasti existujících grafových knihoven pro iOS, nalézt v nich špatné a dobré vlastnosti, a poté vytvořit knihovnu novou, která doplní stávající řešení a poskytne kvalitní zpracování grafů. Součástí je také popis architektury knihovny a náznak konkrétní implementace některých vybraných částí. V závěru práce poskytuje shrnutí dosažených výsledků a náměty na další postup vývoje knihovny. Tato knihovna je implementována pro firmu Inmite s.r.o. a bude sloužit výhradně k jejím interním potřebám.

Klíčová slova: grafy, knihovna, iOS, Inmite, Objective-C, Cocoa Touch, Core Animation, CoreGraphics

Abstract

iOS platform is one of the leading operating systems in nowadays mobile devices industry. iOS applications very often manipulate and visualize various data. Most often, charts are used for this purpose but unfortunately, there is not many of sufficiently good charting libraries for iOS available now. In this document I will try to capture existing charting libraries, describe their advantages and disadvantages and try to come up with custom solution which will complement the existing ones. I will also describe the basic architecture of the custom charting library and include concrete implementation examples of some of its parts. In the last section of the document I will review all the achieved results and suggest next steps in development of the library. The whole library is built for Inmite s.r.o company and is designated for its internal use only.

Keywords: charts, library, iOS, Inmite, Objective-C, Cocoa Touch, Core Animation, CoreGraphics

Seznam použitých zkratek a symbolů

API	–	Rozhraní programového jazyka
AppStore	–	Proprietární služba společnosti Apple, umožňující kupovat a prodávat iOS aplikace
BSD	–	Softwarová licence umžňující volné šíření zdrojového kódu
Cocoa Touch	–	Základní soubor tříd využívaných pro tvorbu iOS aplikací
Core Animation	–	Soubor tříd využívaných k vytváření animací v iOS aplikacích
CoreGraphics	–	Soubor tříd využívaných k implementaci grafického výstupu v iOS aplikacích
CPU	–	Procesor počítače nebo mobilního zařízení
Framework	–	Soubor tříd a funkcí zabezpečujících určitou funkcionalitu
GPLv3	–	Softwarová licence umožňující volné šíření zdrojového kódu s určitými omezeními
GPU	–	Grafický procesor
i386	–	Architektura operačního systému využívaná pro běh iOS Simulátoru na Mac OS X
IDE	–	Prostředí určené k vývoji aplikací
iOS	–	Oprační systém používaný v zařízeních Apple iPhone, iPod Touch a iPad
kanvas	–	Oblast na displeji zařízení, ve které se zobrazuje grafický výstup
Mac OS X	–	Operační systém počítačů od Apple
MVC	–	Návrhový vzor Model View Controller
OpenGL ES	–	Soubor funkcí umožňující zobrazovat 2D a 3D grafiku
PowerPC	–	Typ architektury procesoru dříve používaného v počítačích Apple
UI	–	Uživatelské rozhraní
VCS	–	Systém pro správu a verzování zdrojových kódů
Xcode	–	IDE pro tvorbu iOS aplikací

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Cílem diplomové práce je prozkoumat existující řešení na poli knihoven vykreslujících grafy pro iOS zařízení – iPhone a iPad, a navrhnout vlastní knihovnu, která bude doplňovat nedostatky knihoven již existujících. Knihovna je implementována pro firmu Inmite s.r.o., která se stane majitelem celého řešení.

Nejprve prozkoumám existující řešení, u nichž se pokusím posoudit jejich klady a zápory. Poté navrhnu vlastní řešení knihovny tak, aby co nejlépe vyvažovalo klady a zápory zmíněných knihoven a zároveň, aby splňovalo požadavky od zadavatele, tedy firmy Inmite s.r.o.

Shrnu také možnosti programového rozhraní iOS a ukážu varianty implementace knihovny při použití různých technologií. Pokusím se rozhodnout, která technologie bude pro implementaci nejvýhodnější a jaké jsou k tomu důvody.

V závěru práce popíšu konkrétní úseky mého řešení a některé zajímavější problémy názorně vysvětlím s použitím ukázky zdrojového kódu. Finální řešení grafové knihovny shrnu a názorně ukážu prostřednictvím testovací iOS aplikace určené pro iPhone a iPad.

2 Dostupné grafové knihovny

Tato část popisuje existující grafové knihovny mnou nalezené v době psaní tohoto textu. Shrnuje jejich výhody a nevýhody a snaží se z nich odvodit požadavky na implementaci vlastní grafové knihovny, která by odstranila nedostatky zmíněných, již dostupných řešení.

Inspirován klady a snahou vyvarovat se záporům, poté navrhnu a implementuji vlastní řešení knihovny.

2.1 CorePlot

CorePlot je open source framework, který lze stáhnout na stránce `http://code.google.com/p/core-plot/`. Na této adrese rovněž najdeme návod, jak jej přidat do Xcode projektu, ukázku class diagramu struktury frameworku a jiné dokumenty. Bohužel tam nenajdeme mnoho reálných příkladů, jak knihovnu použít v praxi.

CorePlot je docela dobře použitelný jak pro iOS, tak i pro Mac OS X. Je to asi jeden z nejpoužívanějších open source grafových frameworků. Na domovské stránce je přítomen poměrně dlouhý seznam iOS a Mac aplikací, které ho využívají. Dostupný je také užitečný návod, který obsahuje shrnutí architektury frameworku a vysvětlení, jak se grafy sestavují a ovládají. Tento návod však zahrnuje jen návrh frameworku z vyšší perspektivy, nezachází už do přílišných detailů, takže při vytváření konkrétních grafů je programátor odkázán na procházení zdrojových kódů a odhadování možností grafu podle implementace jeho tříd.

Mezí hlavní nedostatky knihovnknihovny kromě nedostatečné dokumentace, patří příliš velká složitost struktury grafů. Každý graf vyžaduje zbytečné množství dodatečných komponent. I když tyto komponenty programátor ve skutečnosti nepotřebuje využít, přesto je musí vytvořit a sestavit graf tak, aby se správně zobrazil. A to není úplně triviální záležitost. Další špatnou vlastností je ne vždy pěkný výstup. Některé grafy jsou mírně rozmazané a celkově výsledek nepůsobí graficky úhledně. Ve frameworku chybí optimalizace výstupu tak, aby byl pokud možno co nejlepší a neobsahoval rozmazané části. Zároveň firma Inmite, která nyní tuto knihovnu používá v některých svých aplikacích, vyjádřila také nespokojenost s rychlostí vykreslování grafů.

Framework obsahuje asi 10 různých typů grafů.

2.1.1 Výhody

- nabízí výběr z velkého množství typů grafů
- framework je docela rozšířený a je v současnosti aktivně udržován komunitou
- zdrojový kód je pod New BSD licencí, tudíž lze framework použít bez nutnosti dodat k výsledné aplikaci zdrojové kódy
- možnost použití pro iOS i pro Mac OS X

2.1.2 Nevýhody

- složitá struktura, která znesnadňuje jednoduché vytvoření grafu – je potřeba inicializovat mnoho objektů/kontejnerů, aby bylo možné vykreslit jednoduchý graf
- pomalé renderování výstupu
- žádná dostupná oficiální dokumentace
- výstup není graficky pěkný (občas rozmazaný)
- nefunguje spolehlivě (při testování knihovny graf vyhazoval výjimku při své inicializaci a bylo zjevné, že chyba je ve frameworku, nikoliv v aplikaci, která jej používá)

2.1.3 Ukázka zdrojového kódu

Ukázka vytvoření jednoduchého grafu je obsažena v sekci A, jako příloha A.1.

2.2 PowerPlot

PowerPlot je další z open source frameworků. Kvalitou o něco převyšuje dříve zmiňovaný CorePlot. Je dostupný na adrese `http://powerplot.nua-schroers.de/`. Na webu také nalezneme jednoduchý tutoriál, popis struktury frameworku a odkaz na dokumentaci tříd vygenerovanou pomocí nástroje Doxygen. Webové stránky rovněž obsahují jednoduchý přehled novinek, týkajících se této knihovny, ze kterých lze usoudit, že knihovna je stále aktivně vyvíjena autorem. Ovšem je trošku škoda, že zdrojové kódy nejsou dostupné na žádném veřejném VCS úložišti jako například github nebo google code. Takto je knihovna vyvíjena čistě jedním autorem jako tzv. „one man show“ projekt, bez jakékoliv komunity příznivců. Rizikem takového způsobu vývoje je fakt, že pokud autor z nějakého důvodu přestane mít čas na vývoj knihovny, může projekt zcela zaniknout.

Zatím se však zdá, že autor odvádí dobrou práci. Při pohledu do zdrojových kódů frameworku jsem byl téměř okamžitě schopen identifikovat známé návrhové vzory a poměrně snadno jsem se v kódu dokázal zorientovat. Třídy jsou přehledně rozděleny do skupin a obsahují také hojné množství komentářů, díky kterým lze velmi rychle pochopit, jak se která metoda chová a jakým způsobem ji lze použít. Struktura tříd grafů na mě však působila dojmem, že pokud by programátor potřeboval vytvořit vlastní typ grafu, který v knihovně obsažen není, vyžadovalo by to od něj velké množství úsilí.

Velkým přínosem u této knihovny je dobrá dokumentace, která je dokonce lepší než u knihovny CorePlot. Kromě popisu základní struktury objektového návrhu jsou na webové stránce projektu dostupné také ukázky a návody, jak vytvořit jednotlivé grafy. Tato dokumentace poskytuje naprosto dostačující podklady pro začátky používání knihovny a pro pokročilejší úpravy může programátor využít komentářů ve zdrojových kódech, které jsou přítomny skoro u každé metody a třídy.

Knihovna je vydaná pod licencí GPLv3, která říká, že při používání knihovny ve svých zdrojových kódech musí autor spolu s binární verzí aplikace uvolnit i její zdrojové kódy. To však není moc vhodné pro tvorbu iPhone aplikací a distribuci přes AppStore. Autor knihovny však nabízí možnost zakoupit jinou licenci, díky které nebude třeba spolu s aplikací přikládat i její zdrojové kódy.

Framework obsahuje asi 8 různých grafů nabízejících široké možnosti přizpůsobení. Grafy nativně podporují animace.

2.2.1 Výhody

- vytvoření jednoduchého grafu je snadné
- dobrý objektový návrh tříd, využívání známých návrhových vzorů
- velmi kvalitní a udržovaná dokumentace
- probíhá aktivní vývoj knihovny
- široké možnosti přizpůsobení vzhledu grafů
- nativní podpora animací

2.2.2 Nevýhody

- obtížnější možnost sestavení vlastního grafu odvozeného ze základních komponent v knihovně
- některé vlastnosti grafu jsou závislé na pořadí, v jakém jsou nastavovány (např. pokud nastavím text titulku grafu a až poté jeho font, tak se už font nebere v úvahu. Pokud chci změnit font titulku, musím ho nastavit dříve, než samotný text titulku.)
- výstup není vždy ideální – např. rozmazaný text v názvech os
- zdrojové kódy jsou pod licencí GPLv3, která přikazuje distribuovat s aplikací také její zdrojové kódy. Jiný typ licence je zpoplatněný.

2.2.3 Ukázka zdrojového kódu

Ukázka vytvoření jednoduchého grafu je obsažena v sekci A, jako příloha A.2.

2.3 iPhone Charting Library

Jako zástupce placených knihoven jsem vybral knihovnu iPhone Charting Library, která je dostupná jak pro iPhone, tak pro iPad. Knihovna je ke stáhnutí na stránce `http://www.iphonechart.com/`, případně na stránce `http://www.keepedge.com/products/iphone_charting/`, kde nalezneme i několik ukázek použití jednotlivých grafů. Licence stojí $ 999 pro jednu aplikaci.

Na webových stránkách nalezneme sekci novinek z vývoje frameworku. Nejaktuálnější novinka je z června 2010 a zmiňuje vydání verze 1.0. Od té doby se asi nic neudálo, proto soudím, že framework už není aktivně vyvíjen a jeho zdrojové kódy jsou téměř 2 roky zastaralé. Takže už nemusí být zaručeno, že budou stoprocentně funkční i v dnešní době. Další nepříjemností je licenční poplatek, který je nejenže poměrně vysoký, ale navíc je vyžadován zvlášť pro iPhone verzi knihovny a iPad verzi knihovny. Pokud chce programátor využívat grafy v iPhone i iPad aplikaci, musí si zakoupit licence dvě.

Framework (jeho verze ke stažení zdarma) je dostupný pouze jako binární sdílená knihovna (soubor s příponou .a), ale v této knihovně není zkompilovaná architektura i386, pouze PowerPC, což znemožňuje spouštění na iOS Simulátoru a kvůli tomu je aplikaci možné testovat pouze na reálném zařízení. Tento fakt značně znepříjemňuje a hlavně zpomaluje vývoj při použití této knihovny. Protože spolu s knihovnou programátor nezíská zdrojové kódy, je nucen při používání knihovny spoléhat jen na dokumentaci přítomnou na jejím webu. Naštěstí tam však nalezneme příklady pro vytvoření snad všech druhů grafů, které knihovna nabízí, takže ji lze relativně úspěšně používat. Toto řešení ale nepovažuji za naprosto ideální.

Při pohledu na ukázky zdrojových kódů z webu lze usoudit, že struktura knihovny je velmi jednoduchá. Každý graf je samostatná třída, která se nakonfiguruje pomocí svých parametrů a poté se nechá zobrazit. Samotné zobrazení je možné pouze prostřednictvím grafického kontextu (objekt `CGContextRef`, bude zmíněn níže v textu). Grafický kontext je přístupný jen v určitých okamžicích a vykreslování přímo pomocí něj je velmi limitující. V této knihovně by například bylo velmi obtížné vytvářet animace grafů. Sama o sobě knihovna animace nepodporuje.

Na webové stránce projektu je vypsána více než dvacítka různých typů grafů, které framework podporuje. Ke všem jsou také velmi detailní příklady jejich použití.

2.3.1 Výhody

- velká nabídka rozdílných typů grafů
- k dispozici je velmi mnoho ukázkových příkladů, které mohou usnadnit vytváření grafů
- snadné a rychlé vytvoření jednoduchého grafu
- grafický výstup je docela pěkný, až na rozmístění legendy grafu, která vypadá dost rozházeně

2.3.2 Nevýhody

- nestandardní rozhraní tříd – kód není příliš podle zvyklostí jazyka Objective-C
- nedostupnost zdrojových kódů a zastaralost distribuované binární podoby knihovny
- grafy je potřeba sestavit a vykreslit přímo v metodě `drawRect:` předáním objektu `CGContextRef`, což je velmi limitující, například pro vytváření animací grafu

- není dostupná žádná dokumentace objektové struktury frameworku, takže je programátor odkázán jen na ukázkové příklady, které však nejsou vše vysvětlující
- nefunguje stoprocentně – pokoušel jsem se vytvořit koláčový graf, ale ten se bohužel z neznámých důvodu nevykreslil. Tento graf nefunguje ani v oficiální ukázkové aplikaci demonstrující vlastnosti frameworku, která je přiložena v balíčku spolu s frameworkem. Ostatní grafy vypadají, že se vykreslují správně.
- použití frameworku je zpoplatněno licenčním poplatkem, je vyžadována zvlášť licence pro iPhone a pro iPad

2.3.3 Ukázka zdrojového kódu

Ukázka vytvoření jednoduchého grafu je obsažena v sekci A jako příloha A.3.

2.4 Ostatní knihovny

V této části zmíním některé další knihovny, na které jsem narazil. Tyto knihovny dnes už nejsou příliš použitelné pro reálný vývoj aplikací buď to z důvodu jejich zastaralosti nebo proto, že obsahují velmi malé množství grafů a velmi omezené možnosti práce s nimi.

2.4.1 GraphX

Grafový framework sestavený pro Mac OS X aplikace. Domovská stránka projektu je `http://blog.oofn.net/projects/graphx/`. Ačkoliv obrázky na webu vypadají docela slibně, framework je v dnešní době velmi obtížně použitelný. Je totiž implementovaný pro velmi starou verzi Mac OS X primárně ještě pro architekturu PowerPC, která se už několik let nepoužívá.

Zdrojové kódy po menších úpravách šly zkompilovat, tím však práce a úpravy na nich nekončí. Framework není uzpůsobený pro použití na iOS a obsahuje třídy, které jsou dostupné jen pro Mac OS X. Pokud by chtěl programátor tuto knihovnu používat, vyžadovalo by to větší časovou investici na úpravu zdrojových kódů, aby vůbec šly zkompilovat pro iOS a začít používat.

Vzhledem k tomu, že tato knihovna obsahuje jen 3 typy grafů (histogram, korelační diagram a spojnicový graf), nedá se mluvit o reálném použití tohoto frameworku.

2.4.2 iVisualization

Na webové stránce projektu (`http://www.ivisualization.com/`) autoři tvrdí, že se jedná o knihovnu, která nabízí téměř jakýkoliv 2D a 3D graf. Je zde vypsáno také několik dobře znějících vlastností knihovny. Avšak ke stažení je dostupná jen knihovna obsahující koláčový graf. Navíc je knihovna dostupná pouze v binární podobně, zkompilovaná pro iOS 4.2. Přitom v době psaní toto textu je aktuální verze iOS 5.1. Tedy knihovna opět není moc použitelná. Úsměvný je také fakt, že spolu s knihovnou je přiložen dvoustránkový dokument obsahující licenční text o tom, jak smí být koláčový graf používán.

Koláčový graf je údajně jen ukázkou části knihovny, ale na webu není žádná další zmínka o tom, kde je možné získat kompletní knihovnu, ani jaké jsou podmínky k jejímu používání. Dokonce tam ani není zmíněno, kdo je autorem koláčového grafu.

2.4.3 SM2DGraphView

Opět framework původně sestavený pro platformu Mac OS X. Prezentován na webové stránce `http://developer.snowmintcs.com/frameworks/sm2dgraphview`. Bohužel tato knihovna je také velmi stará. Poslední verze Mac OS X, pro kterou byla sestavena, je verze 10.5, a ta už se více než 2 roky nepoužívá. V době psaní tohoto textu má nejnovější verze pořadové číslo 10.7. Zároveň je knihovna dostupná pouze pro Mac OS X a vyžadovala by tedy značného přepisování, aby šla použít pro iOS. Navíc obsahuje jen 3 typy grafů – koláčový, spojnicový a sloupcový, takže není moc o co stát.

3 Možnosti a varianty implementace

V této části se zaměřím na nejčastější návrhové vzory, používané při vývoji iPhone aplikací a porovnám jejich použitelnost pro implementaci mé knihovny. Také ukážu, jaké možnosti grafického výstupu iOS nabízí, jak se vytváří různé typy animací a naznačím zvyklosti při psaní zdrojového kódu v jazyku Objective-C.

Záměrně jsem se snažil v knihovně použít co nejvíce návrhových vzorů a rozdělit kód na logicky spolu související části. Kód je rozdělen do oblastí podle své funkce. Je to například kód, který zajišťuje vykreslování obrazců na grafický kanvas, kód pro počítání matematických výpočtů, reagování na dotyková gesta, a v neposlední řadě třídy obsluhující samotné grafy. Grafy využívají návrhové vzory pro práci se svými daty, zachytávání dotykových gest, kompozici více grafů do jednoho celku a další.

Důkladněji tyto možnosti rozeberu a zhodnotím v následujících podkapitolách.

3.1 Návrhové vzory typické pro iOS

3.1.1 Delegát jako zdroj dat

Delegát je jedním z nejvíce využívaných návrhových vzorů v celém iOS, a proto jsem se rozhodl zakomponovat ho i do své grafové knihovny.

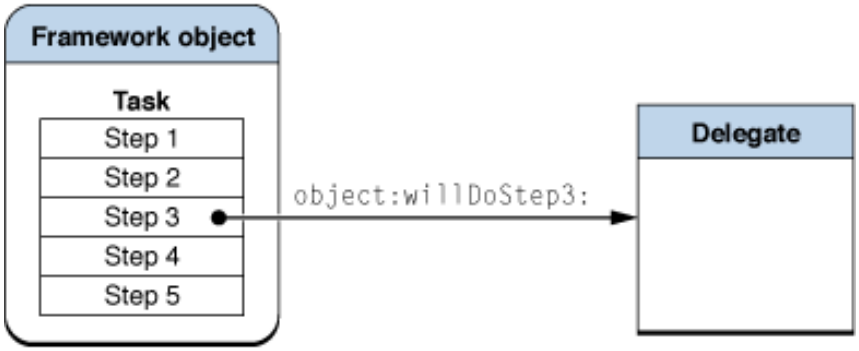
Tento návrhový vzor je mírnou napodobeninou známého vzoru Observer a slouží k poskytování notifikací objektům, které chtějí reagovat na události ve sledovaném objektu.

Princip je následující:

- sledovaný objekt definuje delegáta, který musí implementovat určité metody oznamující, že se udála nějaká událost
- jiný objekt může implementovat tyto metody a sám sebe přiřadit jako delegáta ke sledovanému objektu
- až nastane konkrétní událost, sledovaný objekt zavolá metodu svého delegáta (jež je vlastně objekt, který chce zachytit tuto událost), tudíž se zavolá jeho metoda a objekt v ní může udělat, co uzná za vhodné. Sledovaný objekt předá instanci sama sebe jako parametr metody delegáta.

Toto chování lze znázornit na následujícím diagramu 1.

Velmi typickým použitím delegáta je zprostředkování přístupu k datům objektu. Tento přístup využívá také jedna z nejpoužívanějších komponent Cocoa Touch – `UITableView`, proto jsem si z této třídy vzal inspiraci a způsob předávání dat jsem napodobil. V mém řešení graf nemá veřejné proměnné, obsahující data, ale má objekt delegáta, kterého se vždy na data dotáže. Tento princip používá také existující knihovna CorePlot pro předávání dat grafům.



Obrázek 1: Objekt Framework posílá zprávu svému delegátovi[1]

3.1.2 Datové série a předávání dat v jednom objektu

Další možností, jak předávat data grafům, je vytvořit speciální objekt, který ponese všechny hodnoty, které má graf obsahovat. Tímto způsobem je například řešeno předávání dat v knihovně PowerPlot. Princip je takový, že uživatel grafu nejprve vytvoří pole typu klíč ⇒ hodnota, které poté předá grafu a ten toto pole zobrazí v požadovaném vzhledu. Toto řešení má však své nevýhody.

V první řadě je to fakt, že musíme najednou vytvořit všechny hodnoty a uchovávat je v objektu, což naštěstí v případě grafu, který obsahuje pouze čísla, není takový problém. Tento problém by byl horší, kdyby graf potřeboval jako své hodnoty objekty, které jsou velké a náročné na paměť. Z tohoto důvodu například komponenta `UITableView` používá delegáta pro specifikaci řádků tabulky, protože by to jednoduše bylo příliš náročné, uchovávat všechny řádky jako pole objektů.

Druhou výhodou delegáta oproti poli objektů je, že díky delegátu můžeme zobrazit libovolné a nekonečné množství dat. Již zmiňovaný `UITableView` se například vždy dotáže jen na řádky, které jsou aktuálně viditelné. Nepotřebuje mít informaci o všech řádcích, protože většinu z nich stejně zobrazit nemůže. Stejně tak grafy by mohly zobrazit nekonečná data nebo velmi dlouhou historii dat jednoduše dotázáním se delegáta zajišťujícího data na pouhou podmnožinu dat. Teoreticky by neexistoval rozdíl mezi zobrazením desítek hodnot nebo tisíců hodnot. V případě uchovávání velkého množství hodnot jako pole objektů by to byl obtížně řešitelný problém.

3.1.3 View Controller versus View

U iOS aplikací je samozřejmostí používat návrhový vzor MVC, tedy oddělovat logiku, data a způsob jejich zobrazení. U grafů se nabízí dvojí použití. Buď'to může graf vystupovat jako view controller a zabezpečovat celou logiku ovládání, nebo může být jen jako obyčejný pohled, který se nestará o logiku a dotykové události, ale jen vykreslí data, která má k dispozici.

[1]Zdroj: `http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CocoaFundamentals/Art/delegation.gif`

Z důvodu jednoduchosti jsem se rozhodl pro použití grafu jako obyčejného pohledu. Díky tomu mohou do hry vstupovat další objekty a jednodušeji s grafem spolupracovat, než kdyby to byl view controller. Zároveň jsem si ponechal možnost vytvořit view controller zahrnující více grafů a uzpůsobit ho pro konkrétní analýzy dat a jiná předpřipravená řešení.

3.1.3.1 Zobrazovací cyklus View Controlleru View controller v iOS typicky prochází určitým cyklem při svém vykreslování. Základní třída, ze které všechny view controllery dědí, je `UIViewController`.

1. nejprve dochází k inicializaci view controlleru, metodou `init` nebo `initWithNibName:bundle:`
2. poté se zavolá metoda `viewWillLoad` oznamující, že se začíná inicializovat pohled, který má view controller zobrazit
3. pohled se vytváří metodou `loadView`. V této metodě buď'to programátor pohled sám inicializuje nebo pokud k vytváření pohledu použil nástroj na tvorbu uživatelského rozhraní – Interface Builder – tak využije toho, že pohled se vytvoří sám v implementaci `loadView` metody v mateřské třídě, tudíž se o jeho vytvoření sám starat nemusí
4. po vytvoření pohledu se zavolá metoda `viewDidLoad`, ve které může programátor pohled dodatečně upravit podle potřeby. Této metody se využívá zejména, pokud jsou pohledy tvořeny v Interface Builderu
5. při zobrazování view controlleru v aplikaci se zavolá metoda `viewWillAppear:`, která oznamuje, že pohled se právě začíná zobrazovat
6. po zobrazení celého pohledu se zavolá metoda `viewDidAppear:` naznačující, že zobrazení bylo ukončeno

Toto jsou metody první poloviny zobrazovacího cyklu view controlleru – zobrazení. Každá z těchto metod má svůj význam a programátor by měl vědět, které kroky se v jaké metodě mají provádět. Například metoda `viewDidLoad` se zavolá pouze při úplně prvním vytváření pohledu pro view controller, ale už se nevolá při přepínání mezi view controllery, protože view controller zůstává inicializován i se svým pohledem. Proto pokud programátor chce při každém zobrazeni controlleru zajistit vždy aktuální data, musí svůj kód pro aktualizaci dat vložit do metody `viewWillAppear:`, která se volá při každém zobrazení view controlleru v aplikaci.

Druhá polovina zobrazovacího cyklu – skrytí view controlleru – je analogická k té první. View controller volá metody v následujícím pořadí:
`viewWillDisappear:`, `viewDidDisappear:`, `viewWillUnload:`, `viewDidUnload:`, `dealloc`.

První dvě metody se volají před každým přepnutím na jiný view controller. V nich je například vhodné uložit změny, které se v pohledu a datech provedly. Druhé dvě metody se volají pouze jednou, ve chvíli, kdy se má celý view controller odstranit z paměti, což často bývá až při vypnutí celé aplikace. Konečně metoda `dealloc` zajistí samotné zrušení objektu view controlleru a uvolnění paměti.

3.1.3.2 Pohled Pohled, tedy třída zděděná z `UIView`, je oproti view controlleru velmi jednoduchý. Zobrazuje pouze informace, které má k dispozici. Jeho významnou vlastností je, že v sobě dokáže zakomponovat další pohledy a tedy vytvářet komplexnější soubor, vytvořený z více pohledů najednou. Takto například funguje třída `UITableView` používaná pro vytváření tabulek. Zobrazuje jednotlivé řádky v tabulce, které jsou také pohledy a každý z řádků rovněž obsahuje své pohledy, kterými mohou být texty, tlačítka nebo obrázky. Pohled, který zobrazuje view controller, tedy často bývá velmi komplexním složením několika menších pohledů. Samozřejmě čím složitější pohled je, tím náročnější je jeho zobrazení. Proto se některé techniky optimalizace výkonnosti aplikace zabývají právě snížením počtu vnořených pohledů.

Cyklus zobrazování pohledu je následující:

1. inicializace pohledu a předání obdélníku udávajícího rozměr, ve kterém se pohled má zobrazit
2. zavolání metody `layoutSubviews`, která zajistí rozmístění případných sub-pohledů, které pohled obsahuje
3. automatické zavolání metody `drawRect:` zajišťující vykreslení pohledu

Pokud se data pro pohled změní a je třeba vykreslit je znova, stačí zavolat metodu `setNeedsDisplay` nebo `setNeedsDisplayInRect:` a pohled se sám překreslí, až přijde ta správná chvíle. Toto je důležité – až přijde ta správná chvíle. Pohled není možné vykreslit kdykoliv si programátor zamane. Tohle je častou chybou začínajících programátorů, kteří se snaží vykreslovat pohled například v metodách zachycujících dotykové události. Správný postup při zachycení události je pouze změnit vlastnosti pohledu a právě pomocí metody `setNeedsDisplay` mu oznámit, že se má překreslit. iOS pravidelně takovéto změny sleduje a pohled, který má nastaveno, že se má překreslit, bude automaticky překreslen. Obecně (s malými výjimkami) platí, že vykreslování by se mělo dít jen a pouze v metodě `drawRect:`. Metoda bude zavolána na vyžádání frameworku Cocoa Touch ve chvíli, kdy bude vykreslení potřeba. Programátor by ji nikdy neměl sám volat v kódu.

3.2 Způsoby grafického výstupu pro iOS

Při diskuzi o pohledech a jejich zobrazování jsem už nakousl otázku samotného vykreslování pomocí grafického enginu. iOS nabízí několik možností vykreslování. Pokusím se je teď krátce rozebrat jednu po druhé a přiblížit možnosti jejich použití pro účely vytváření grafů.

3.2.1 Graf složený ze spousty pohledů

Jedním z možných řešení je vytvořit graf jako složení více pohledů. Nejjednodušeji se toto řešení dá představit na sloupcovém grafu. Každý sloupeček by v tomto případě byl samostatný pohled.

Na příkladu sloupcového grafu toto řešení vypadá (a dost pravděpodobně i je) velmi jednoduché a rychlé. Pro každou hodnotu v grafu stačí rozhodnout výšku příslušného sloupce, vytvořit obdélník o této výšce a přidat ho jako sub-pohled na správnou pozici. Navíc v takto sestaveném grafu by se také velmi jednoduše realizovaly animace, protože Cocoa Touch umožňuje programátorsky velmi snadno vytvářet animace pro změnu velikosti, průhlednosti či rotaci celých pohledů.

Problém by však mohl nastat u grafů, které mají velký počet hodnot. Tehdy by graf obsahoval velký počet sub-pohledů. Jak bylo již zmíněno, platí: čím větší počet sub-pohledů, tím delší čas potřebný pro jejich vykreslení.

U ostatních typů grafů by mohla nastat rozdílná situace. Třeba spojnicový graf je jen křivka, která se různě klikatí napříč stupnicí hodnot. Nevýhodou v této situaci je fakt, že pohled se v základu vykresluje jako obdélník, ale spojnicový graf potřebuje vykreslit ne zrovna pravidelnou křivku či lomenou čáru. Musela by se proto vytvořit nějaká čára, podle které by se pohled seřízl a teprve pak přidal do grafu. Ale právě takovéto definování křivky a seříznutí je velmi náročné a proto je v této situaci výhodnější křivku vykreslit přímo, než ji vytvořit jen aby se podle ní seřízl pohled, který se teprve přidá jako sub-pohled.

3.2.2 CoreGraphics

Na nižší úrovni grafického výstupu je framework CoreGraphics. Ve skutečnosti tento framework používají i pohledy, aby zobrazily samy sebe. CoreGraphics není objektový. Je napsaný jako soubor funkcí z jazyka C, které zabezpečují vykreslování na *grafický kanvas*. Toto vykreslování se děje, jak už jsem zmínil dříve, téměř výhradně v metodě `drawRect:` v jednotlivých pohledech.

Framework CoreGraphics je vhodný pro zobrazení 2D objektů, velmi hojně je využíván programátory a velmi dobře optimalizován na straně iOS, tudíž je velmi rychlý a efektivní.

3.2.2.1 Architektura CoreGraphics Framework CoreGraphics je také někdy nazýván Quartz 2D. Oba tyto názvy z velké části označují to stejné. Základním stavebním kamenem frameworku je grafický kontext – definovaný strukturou `CGContextRef`.

Princip vykreslování funguje tak, že získáme grafický kontext a jeho parametry nějakým způsobem modifikujeme, například přidáním křivky, barvy výplně nebo specifikací tloušťky čáry či fontu. Poté kontext necháme vykreslit. Samotný objekt kontextu reprezentuje místo, kde se výstup má zobrazit. Nejčastěji se setkáme s kontextem pro vykreslování na displej telefonu (případně obrazovku u desktopových aplikací), ale může to stejně tak být i PDF kontext, bitmapový kontext nebo kontext pro tisk na tiskárně. Důležité je, že stejný kód je možné vykreslit na různých místech jen změnou kontextu.

Kromě kontextu obsahuje CoreGraphics ještě další datové typy pro uchovávání informací o použitých barvách, křivkách, obrázcích, vzorech, stínech a spoustě dalších. Tyto objekty jsou většinou jen nositeli informace, jakým způsobem kontext upravit a vykreslit co je potřeba.

Kontext v sobě dokáže uchovávat několik grafických stavů. Vykresluje se vždy **aktuální grafický stav**. Chceme-li tedy nastavit barvu a tloušťku čáry vykreslené kontextem, použijeme k tomu funkce, které změní jeho aktuální grafický stav. Rutiny zajišťující vykreslení pak tento aktuální stav použijí pro zobrazení výstupu. Grafický kontext si udržuje zásobník takovýchto stavů. Vytvářením a úpravou stavů lze pak docílit vytvoření velmi komplexního obrazu i za pomocí takto jednoduchého systému.

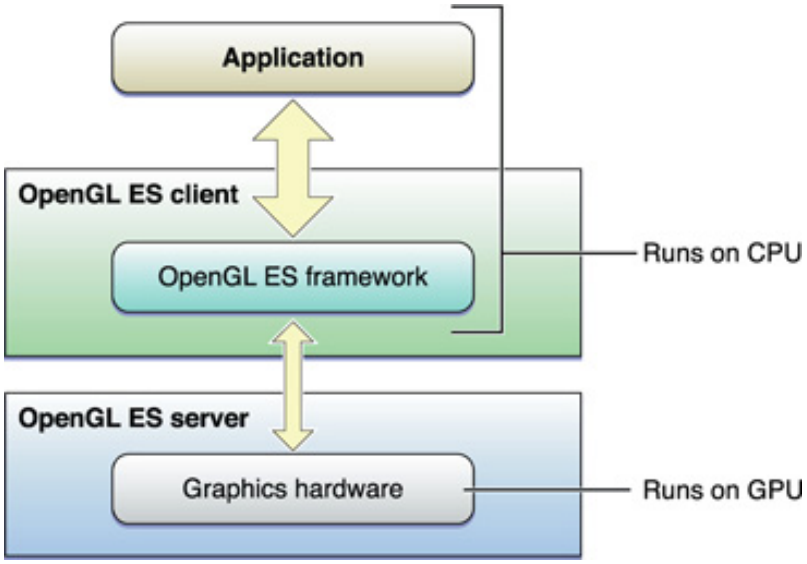
Nový grafický stav se vytváří voláním funkce `CGContextSaveGState`. Po vytvoření se stav automaticky přidá na vrchol zásobníku. Po dokončení vykreslování v tomto stavu musí programátor zavolat funkci `CGContextRestoreGState`, která stav odstraní ze zásobníku a jako aktuální přiřadí ten, který byl uložen pod ním, a upraví parametry grafického kontextu tak, aby odpovídaly aktuálnímu stavu. Je proto velmi důležité ujistit se, že volání těchto dvou metod je balancované, tedy po každém přidání stavu do zásobníku následuje také jeho odebrání. Jinak se vykreslování zcela určitě nepodaří podle našich představ.

Vybrané parametry kontextu, které souvisí s aktuálním grafickým stavem:

- transformační matice aplikovaná na výstup (Current transformation matrix, CTM)
- maska výstupu (Clipping area)
- čára: šířka, zaoblení rohů, zaoblení konce, plnost čáry
- nastavení antialiasingu
- nastavení barvy výplně a okraje
- hodnota průhlednosti (alpha)
- použité barevné schéma
- text: typ a velikost fontu, mezery mezi znaky, mód vykreslování textu
- způsob prolínání výstupu s jinými grafickými stavy

Při počítání souřadnic bodů zobrazovaných pomocí CoreGraphics je potřeba si uvědomit, že počátek pro souřadnice pohledu je v levém horním rohu a vektory os směřují doprava a dolů, kdežto počátek pro souřadnice grafického kontextu je v levém spodním rohu a osy směřují doprava a nahoru. Je tedy nutné y souřadnici vždy správně přepočítat tak, aby výsledný obraz nebyl zrcadlově obrácený.

Popis architektury frameworku CoreGraphics byl převzat z [8].



Obrázek 2: Architektura OpenGL ES pro iOS[2]

3.2.3 OpenGL ES

iOS nabízí také možnost využití zobrazovacího enginu OpenGL ES sloužícího pro zobrazování 2D a 3D scén. Podporováno je OpenGL ES 1.1 a také OpenGL ES 2.0. Oboje jsou multiplatformní soubory funkcí v jazyku C. Samotné funkce OpenGL ES jsou hardwarově akcelerované přímo na grafické kartě, čímž je docíleno velké výkonnosti.

3.2.3.1 Architektura OpenGL ES v iOS OpenGL ES je založeno na architektuře klient-server. Aplikace, využívající OpenGL ES, volá funkce na OpenGL ES klienta. Ten zpracuje volání a pokud je to nutné, pošle funkci dále ve formě příkazu pro OpenGL ES server. Klient zpracovává volání na CPU, server běží na GPU. Rozložení funkcí na klienta a server a komunikace mezi nimi je specifická pro konkrétní implementaci a platformu OpenGL ES a je různá pro iOS a pro Mac OS X.

Standard OpenGL ES nedefinuje spolupráci grafických funkcí se systémem. Každá platforma proto musí poskytnout svoji vlastní implementaci funkcí pro vytváření renderovacích kontextů a systémových framebufferů.

Renderovací kontext je vnitřní stav informací, který OpenGL ES používá. Každá aplikace musí mít svůj vlastní kontext a může jich používat i více.

Systémový framebuffer slouží k uchovávání příkazů pro vykreslování pomocí OpenGL ES a bývá závislý na grafickém systému konkrétního zařízení, na kterém aplikace běží. iOS nepodporuje systémové framebuffery, místo toho framebuffer z OpenGL ES rozši-

[2]Zdroj: `https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Art/cpu_gpu.jpg`

řuje tak, aby sdílel svá data se systémovou vrstvou Core Animation, která je jediným vykreslovacím nástrojem pro iOS.

OpenGL ES je velmi výhodné z hlediska výkonnosti a plynulosti grafických animací a je vhodné především k renderování 3D scén. Pro vykreslování grafů, které v základu budou jen ve 2D prostoru, je použití OpenGL možná příliš předimenzované. OpenGL je dle mého názoru vhodnější pro zobrazení celoobrazovkových scén a jejich animaci. Grafy velmi pravděpodobně budou zobrazovány jen jako sub-pohledy ve view controllerech.

Další nevýhodou je zachycování dotykových událostí v OpenGL. Programátor má přístup pouze k souřadnicím pixelů, kde došlo k dotyku, ale nemá žádné další dodatečné informace. Naproti tomu při vykreslování pomocí CoreGraphics můžeme využít toho, že graf dědí z `UIView` a umožňuje sofistikovanější zachytávání dotyků, např. pomocí tříd zděděných z `UIGestureRecognizer`.

Popis architektury OpenGL ES byl převzat z [9].

3.3 Animace prvků na obrazovce

iOS nabízí široké možnosti animování prvků v aplikacích. V této podkapitole ukážu, jaké jsou možnosti zakomponování animací do iOS aplikace a jaké typy animací jsou vhodné pro grafovou knihovnu.

3.3.1 Animace pomocí změny properties pohledů

Jednou z věcí, které lze díky frameworku Cocoa Touch dělat velmi jednoduše, jsou základní animace v pohledech. Celý princip těchto animací spočívá v tom, že pohledu řekneme jakým způsobem se má animovat, jak dlouho animace má trvat a zbytek necháme na něm. Animace se provede sama a vypadá vždy dobře. Cocoa Touch umožňuje takto animovat jen některé properties třídy `UIView`, které je možno tímto způsobem animovat. Jejich výčet je vypsán v tabulce 1.

V současné době existují dvě možnosti zápisu těchto animací. Ta starší spočívá v nadefinování animace pomocí statických metod třídy `UIView` a můžeme ji rozdělit na tři části.

1. definice vlastností animace – délka trvání, zpoždění začátku, metoda zavolaná při započetí/dokončení animace, kontext animace atp. K tomu slouží například metody (`[UIView beginAnimation:inContext:]`, `[UIView setAnimationDuration:]` a další)

2. definice změn provedených během animace – zde nastavíme properties animovaného pohledu tak, jak je chceme po dokončení animace

3. odsouhlasení animace – každou animaci je potřeba potvrdit pomocí metody `[UIView commitAnimations]`. Tehdy se teprve provede

property	význam
frame	Obdélník, ve kterém se pohled vykresluje. V závislosti na souřadném systému nadřazeného pohledu.
bounds	Obdélník, ve kterém se pohled vykresluje. V závislosti na svém vnitřním systému nadřazeného pohledu.
center	Střed trojúhelníku v závislosti na souřadném systému nadřazeného pohledu.
transform	Afinní transformace aplikovaná na pohled.
alpha	Průhlednost pohledu.
backgroundColor	Barva pozadí pohledu.
contentStretch	Obdélník definující oblast, která se může roztáhnout při změně velikost pohledu (například při změně zobrazovacího módu displeje z landscape na portrait).

Tabulka 1: Vlastnosti `UIView`, které lze animovat

Druhým způsobem, jak takovouto animaci vytvořit, je použití bloků, tedy anonymních funkcí. Tato metoda je ještě jednodušší, avšak ne vždy použitelná, protože starší verze iOS bloky nepodporují.

K tomuto přístupu se opět využívají statické metody třídy `UIView`, například metoda `[UIView animateWithDuration:animations:completion:]`, jejíž konkrétní použití je ukázáno v následujícím zdrojovém kódu:

```
CGPoint originalCenter = icon.center;
[UIView animateWithDuration:2.0
                 animations:^{
                     CGPoint center = icon.center;
                     center.y += 60;
                     icon.center = center;
                 }
                 completion:^(BOOL finished){
                     [UIView animateWithDuration:2.0
                                      animations:^{
                                          icon.center = originalCenter;
                                      }
                                      completion:^(BOOL finished){
                                          ;
                                      }];
                 }];
```

Výpis 1: Animace pohledu pomocí bloků

Tento kód bude během dvou sekund animovat posun ikony o 60 bodů níže a po dokončení této animace se spustí další dvousekundová animace, která ikonu vrátí na její původní místo.

Je jasné, že tímto přístupem nelze docílit úplně všech typů animací. Zvláště některé složitější animace takto vytvořit nelze a musí proto být použit jiný způsob.

3.3.2 Rozsáhlejší animace pomocí Core Animation

Core Animation je další ze základních frameworků zajišťujících zobrazování a animaci objektů na obrazovce. Je také hlavním nástrojem pro tvorbu animací. Předchozí popisované animace, které `UIView` podporuje, interně využívají třídy Core Animation.

Třídy Core Animation se dají rozdělit do několika kategorií:

- třídy reprezentující vrstvy zobrazeného obsahu
- třídy pro animace a jejich časování
- třídy sloužící jako definice rozvržení (layoutu) obsahu
- třídy spojující animace několika vrstev najednou do atomických transakcí

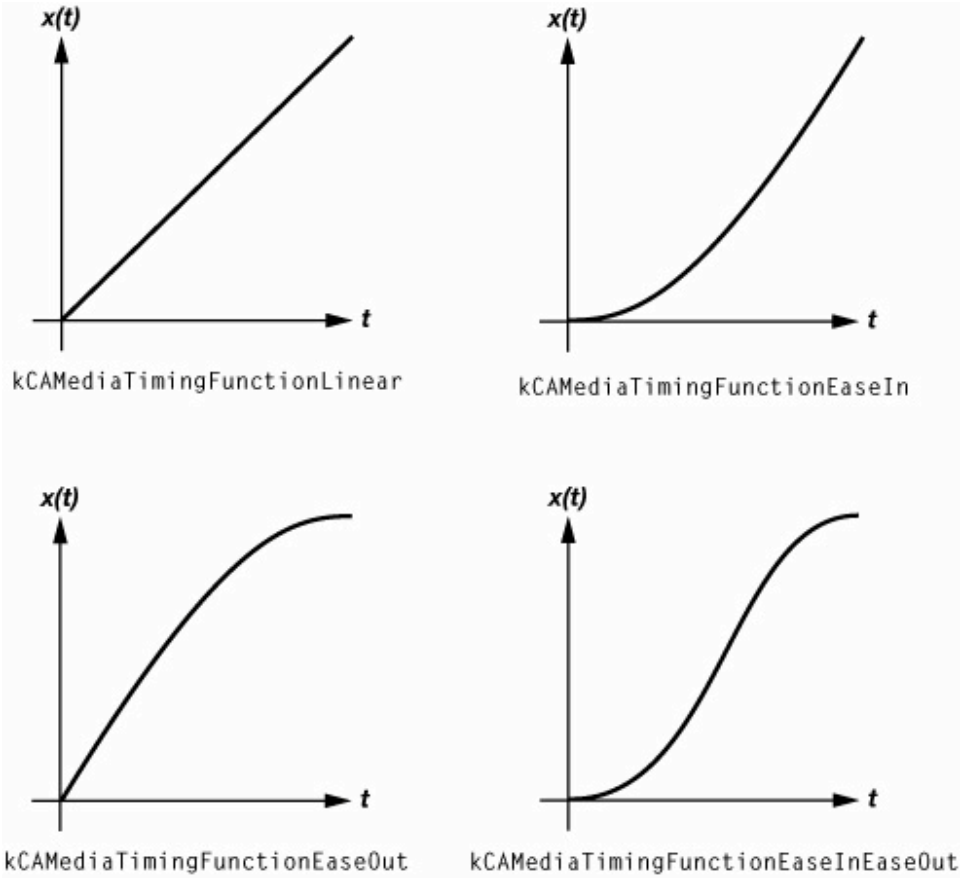
V této části se zaměřím pouze na třídy definující animace a jejich časovací funkce. Jsou to třídy `CAAnimation` plus její potomci, a protokoly `CAMediaTiming` a `CAAction`.

Stejně jako třída `UIView`, také třídy vrstev obsahu lze animovat změnou nastavení jejich properties. Toto platí pro třídu `CALayer` a její potomky. `CALayer` ovšem dokáže animovat mnohem více vlastností, než tomu bylo u `UIView`. Core Animation framework umožňuje animovat buď to vybrané atributy vrstev nebo obsah celé vrstvy najednou. Používá k tomu základní animace vytvářené změnou některých properties vrstev nebo tzv. *key-frame* animace definující průběh krok za krokem. Protokol `CAMediaTiming` určuje časování, rychlost, průběh a počet opakování animace. Vrstvy také dokáží vyvolat animaci jako odpověď na události zachycené ve vrstvě. K tomu slouží již zmiňovaný protokol `CAAction`.

Z hlediska časování animací nabízí Core Animation už předpřipravené časovací funkce – s lineárním průběhem, zpomalené nebo zrychlené v čase podle definované Beziérovy křivky a několik dalších. Samozřejmě existuje možnost definovat si svoji vlastní časovací funkci.

Pro vytvoření animace je nejdříve potřeba vytvořit instanci třídy dědící z `CAAnimation`, často se používá třída `CABasicAnimation`. Této instanci nadefinujeme, kterou property vrstvy má animovat, dobu trvání animace, časový průběh, počet opakování a další. Takto vytvořenou animaci přiřadíme libovolné vrstvě, tedy instanci třídy `CALayer`, pomocí metody `addAnimation:forKey:`. Animace se automaticky provede. Celý princip ilustruje následující ukázkový kód:

```
CABasicAnimation *hideAnimation;

hideAnimation =[CABasicAnimation animationWithKeyPath:@"opacity"];
hideAnimation.fromValue=[NSNumber numberWithFloat:1.0];
hideAnimation.toValue=[NSNumber numberWithFloat:0.0];
hideAnimation.timingFunction = [CAMediaTimingFunction functionWithName:
    kCAMediaTimingFunctionEaseInEaseOut];
hideAnimation.duration=2.5;
hideAnimation.repeatCount=5;
hideAnimation.autoreverses=YES;
```



Obrázek 3: Průběh základních funkcí pro časování animací[3]

```
[theLayer addAnimation:hideAnimation forKey:@"animateOpacity"];
```

Výpis 2: Ukázka základní animace `CAAnimation`

O něco náročnější jsou key-frame animace. Třída `CAKeyframeAnimation`. Tento druh animací jsem se snažil využít pro animování točení koláčového grafu, při simulaci magnetického tlumeného kmitání kolem středu výseče. Ovšem jak jsem nakonec zjistil, key-frame animace nebyly pro mé účely dostačující. Požadovaný efekt sice bylo možné realizovat, ale působil příliš uměle a nepřirozeně. Přestože jsem se snažil nastavit kroky animace a její časování tak, aby co nejvěrněji simulovaly reálné chování, nepodařilo se mi docílit uspokojivého výsledku.

Krokovací animace se vytváří obdobným způsobem jako základní animace, jen s tím rozdílem, že místo hodnoty na počátku a na konci je potřeba nadefinovat sérii hodnot a k ní příslušící sérii časovacích funkcí pro každý krok animace. Dále je nutno mít na paměti, že počet hodnot v sérii musí být roven počtu kroků animace, kdežto počet časovacích funkcí je o 1 menší – každá časovací funkce totiž definuje průběh od jednoho kroku k dalšímu.

Po vytvoření key-frame animace ji připojíme k libovolné vrstvě opět využitím metody `addAnimation:forKey:`, jako kteroukoliv jinou animaci.

3.3.3 Vlastní implementace animací

Poslední možností, jak docílit animovaného obsahu, kterou zde popíši, je implementace animace úplně od nuly. Tento způsob je někdy nezbytný, hlavně pro složité animace, které nelze vytvořit žádným z předchozích způsobů. Nebo pro animace, které potřebují precizní časování a průběh.

Princip spočívá v tom, že nastavíme časovač, který bude pravidelně v požadovaném intervalu volat metodu, která způsobí překreslení pohledu. Pokud například požadujeme animaci o rychlosti 40 oken za sekundu, musíme časovač nastavit tak, aby metodu opakovaně volal s periodou $1/40$ sec.

V metodě pak definujeme cokoliv je potřeba udělat, aby mohl nastat další krok animace a pohled necháme překreslit pomocí metody `setNeedsDisplay`. Pamatujme na to, že v této metodě se nesmíme pokoušet o vlastní kreslení, protože ještě není ten správný čas. Viz předchozí část textu 3.2.2 o CoreGraphics.

Tento styl animace je v mé knihovně využíván pro překreslování některých grafů. V metodě volané v každém kroku se změní některé properties grafu a zavolá se na něj metoda `setNeedsDisplay`, díky které, až přijde čas vykreslování, se zavolá metoda `drawRect:`, ta podle aktuálního nastavení properties graf vykreslí v požadované podobě.

Ačkoliv toto řešení může na první pohled vypadat jako velmi neefektivní, funguje naprosto plynule. I na slabším hardware (iPhone 3GS) a při rychlosti 60 oken za sekundu.

Konkrétní ukázka tohoto přístupu k animacím bude uvedena v části 5.2.2, zabývající se implementací rotace koláčového grafu.

[3] Zdroj: `https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Art/cpu_gpu.jpg`

4 Požadavky na vlastní řešení

4.1 Požadavky zadavatele – Inmite s.r.o.

Postup formulace požadavků na grafovou knihovnu a funkcí, kterými bude disponovat, s firmou Inmite s.r.o., byl velmi agilní. Před započetím implementace byly dohodnuty jen opravdu základní požadavky na některé typy interakcí a jen velmi málo typů grafů. Bylo to dáno hlavně tím, že firma Inmite všechny své projekty řeší agilním způsobem a tedy požadavky vznikají až v průběhu vývoje produktu. Druhým důvodem bylo, že firma Inmite při započetí vývoje ve skutečnosti potřebovala do svých aplikací jen tři typy grafů, a proto neměla přesnou představu, jaké další typy grafů v budoucnu bude využívat.

Pro referenční příklad k vizualizaci grafů mi sloužila aplikace Roambi Analytics Visualizer[4] a mým cílem bylo co nejvíce funkčně napodobit některé grafy obsažené v této aplikaci.

4.1.1 Základní požadavky frameworku

Mezi základní požadavky frameworku jsem zařadil ty, jež jsme dohodli společně s firmou Inmite s.r.o. Požadavky jsou rozděleny do kategorií na funkční, nefunkční a na popis funkcí a chování jednotlivých typů grafů.

4.1.1.1 Funkční

- Možnost interakce s grafy pomocí dotykových událostí
- Plynulé animace grafů (minimálně 40 fps)
- Možnost zobrazení grafu jak na iPhone, tak na iPadu
- Export grafu do PNG či JPG obrázku

4.1.1.2 Nefunkční

- *iOS like* styl požívání frameworku – využití delegátů a data sourců (podobný styl, na kterém je založena standardní komponenta `UITableView`)
- Pěkně vypadající grafy
- Uspokojivá rychlost vykreslování
- Jednoduchý způsob používání grafů a jejich úpravy

[4] `http://itunes.apple.com/us/app/roambi-analytics-visualizer/id315020789`

4.1.1.3 Požadované typy grafů [5]

1. koláčový
2. spojnicový s výplní
3. sloupcový

4.1.1.4 Obecné požadavky grafů Měla by být možnost zobrazit více souborů hodnot v jednom typu grafu. Tzn. ve spojnicovém grafu více křivek pro různé série dat, ve sloupcovém grafu pro každou sérii dat sloupce různých barev, atp.

U os grafu by měla být možnost nastavit si:

Z pohledu vizualizace:

- velmi důležité: jestli jsou osy, major tickery/grid[6] a minor tickery/grid, vidět
- středně důležité: barva osy, major tickeru/gridu a minor tickeru/gridu
- málo důležité: tloušťka čar osy, major tickeru/gridu a minor tickeru/gridu

Z pohledu dat:

- velmi důležité: mít možnost pro každý bod explicitně nastavit popisek osy
- málo důležité: mít možnost nastavit krok pro tickery

4.1.1.5 Koláčový graf Koláčový graf by měl umět zobrazit barevné výseče ve velikosti tak, aby reprezentovaly data grafu předaná. Základní vlastností grafu je možnost otáčení dotykem prstu. Programátor bude moci nastavit velikost grafu, barvu a hodnoty pro každou výseč. Bude mít také možnost nastavit barvu a tloušťku čáry oddělující jednotlivé výseče.

Graf umožní nastavit pozici zobáčku, jenž bude poskytovat informace, která výseč je aktuálně zobáčkem označena a jaká je hodnota v této výseči. Po přerušení tažení koláče bude graf nadále pokračovat v otáčení setrvačným pohybem rychlostí úměrnou té, kterou dotyk skončil. Navíc při dokončení setrvačného otáčení se graf zastaví tak, aby zobáček ukazoval do středu naposledy vybrané kruhové výseče. Animace zarovnání do středu může vypadat tak, že zobáček střed výseče magneticky přitahuje a graf několikrát osciluje, než se ve středu výseče zastaví.

[5] jelikož zadavatel primárně požaduje pouze tři typy grafů, doplním do knihovny některé další dle svého uvážení

[6] tickery jsou dílky na ose, grid je mřížka uvnitř grafu zvýrazňující některé hodnoty

4.1.1.6 Spojnicový graf s výplní Tento typ grafu bude vypadat jako typický spojnicový graf, ale bude umožňovat nastavit výplň oblasti mezi křivkou grafu a osou x. Výplň grafu může být jednolitá barva, barevný přechod nebo libovolný obrázek. Výplň vznikne tak, že se pozadí vykreslí přes celý obdélník grafu a poté se seřízne podle datové křivky, část pozadí pod křivkou se ponechá, část nad křivkou se zobrazovat nebude.

Graf bude reagovat na dotyky a tažení prstu. Při dotyku se v grafu vizuálně zvýrazní hodnota na datové křivce odpovídající pozici dotyku prstu a zobrazí se svislá čára naznačující hodnotu na ose X korespondující se zvýrazněnou hodnotou na křivce. Při tažení prstu se bude měnit vyznačená oblast a svislá čára tak, aby vyznačovaly vždy hodnotu, na kterou ukazuje prst.

4.1.1.7 Sloupcový graf Bude obsahovat sloupce. Sloupce budou animovaně růst. Výplň sloupců bude stejná jako u spojnicového grafu s výplní – barva, barevný přechod nebo obrázek. Graf bude umět automaticky přizpůsobit šířku sloupců jejich počtu a místu vymezenému pro zobrazení grafu. Nereaguje na žádná dotyková gesta

4.2 Dodatečné požadavky na knihovnu

Po prozkoumání a zhodnocení existujících grafových knihoven jsem navrhl vlastní požadavky, které doplňují ty od zadavatelské firmy, a zároveň se snaží vyvarovat nedostatkům existujících knihoven. Definoval jsem také dodatečné typy grafů, které považuji za užitečné a vhodné pro zobrazení na iOS zařízeních. Celkem bude knihovna obsahovat 8 různých typů grafů.

Mezi další vhodné požadavky patří zejména:

- jednoduché vytvoření nového typu grafu použitím základních komponent knihovny
- možnost spojení více grafů do komplexnější studie
- každý graf podporuje animace
- graf se dokáže uspokojivě zobrazit v různých velikostech a poměrech stran

4.2.1 Dodatečné typy grafů

4.2.1.1 Spojnicový graf Typický spojnicový graf, který obsahuje křivku vytvořenou spojením několika bodů. Tento graf je odvozen ze spojnicového s výplní s tím rozdílem, že oblast pod křivkou dat je prázdná, tedy neobsahuje žádnou výplň. Graf dokáže reagovat na stejná dotyková gesta, jako jeho obdoba s výplní.

4.2.1.2 OHLC burzovní graf Tento graf je hojně využíván například pro zobrazování pohybu akcií či komodit a ostatních finančních artiklů. Název OHLC pochází z anglického open, high, low, close a vyjadřuje čtyři hodnoty v určitém časovém úseku – otevírací, nejvyšší, nejnižší a uzavírací.

Rozmezí nejvyšší a nejnižší hodnoty bude graf zobrazovat jako svislou čáru, uzavírací a otevírací hodnoty budou zobrazeny jako krátké vodorovné čáry směřující vlevo (otevírací) a vpravo (uzavírací). Graf bude také podporovat alternativní zobrazení v podobě svíčkového grafu. Toto zobrazení se liší jen tím, že rozmezí otevírací a uzavírací hodnoty je zobrazeno v podobě barevného obdélníku. Zelená barva naznačuje, že otevírací hodnota byla na začátku časového úseku nižší než uzavírací na jeho konci, opačný případ je znázorněn barvou červenou.

Graf bude podporovat animaci zobrazení hodnot, při které hodnoty jakoby vyrostou nahoru a dolů ze svého středu. Nebude reagovat na dotyková gesta.

4.2.1.3 Bublinový graf Tento graf slouží pro zobrazení trojrozměrných dat. Osy x a y reprezentují jakékoliv hodnoty a třetí rozměr je demonstrován velikostí bubliny v grafu. Velikost bubliny je nutné zobrazit jako procentuální hodnotu tedy musí nabývat hodnot z intervalu <0;1>. Graf automaticky určí velikost bublin podle daných procentuálních hodnot a upraví rozmezí os x a y tak, aby dokázalo pojmout i dodatečné navýšení rozměrů bublin. Barvu bublin je možné nastavit libovolně.

Graf bude podporovat animaci, při které se bublina z nulové velikosti zvětší na svůj konečný rozměr. Nebude reagovat na dotyková gesta.

4.2.1.4 Histogram Je jedním ze základních statických grafů. Zobrazuje distribuci dat v určitém intervalu hodnot pomocí sloupcového diagramu. Zároveň dokáže v každém intervalu procentuálně vyjádřit i kumulativní hodnotu četnosti vůči součtu všech hodnot. Graf tedy obsahuje jednak sloupce, ale také křivku jiných hodnot. U každé kumulativní četnosti je vypsána příslušná procentuální hodnota. Graf umožní nastavit výplň sloupců, stejně jako sloupcový diagram, a zvolit libovolnou barvu křivky kumulativních četností.

Při animovaném zobrazení histogramu vyjíždějí sloupce odspoda a zároveň zleva doprava se postupně objevuje křivka zobrazující kumulativní četnost. Těsně před úplným dokončením animace se zobrazí samotné číselné hodnoty kumulativní četnosti. Graf nebude reagovat na dotyková gesta.

4.2.1.5 Bodový graf (korelační diagram) Je dalším z často používaných statistických grafů. Tento graf je charakteristický tím, že dokáže zobrazit více y hodnot příslušících jedné x souřadnici. Používá se na vyjádření míry korelace mezi dvěma statistickými proměnnými. Graf umí takovéto body zobrazit jako malé barevné čtverečky rozmístěné podle své pozice. Graf umožní nastavit libovolnou barvu těmto čtverečkům.

Při animaci se čtverečky budou postupně zobrazovat zleva doprava. Graf nebude reagovat na dotyková gesta.

5 Implementace vlastní grafové knihovny

Tato kapitola slouží k popisu procesu implementace grafové knihovny. Je zde zachyceno mé bádání nad návrhovými vzory, které jsem v knihovně použil a nad způsobem rozdělení zdrojových kódů podle logických závislostí. Dále jsou zde poskytnuty ukázky některých konkrétních problémů při implementaci knihovny a zdůvodnění použitého řešení. Poslední část kapitoly poskytuje obecné povídání o možnostech grafického vykreslování pro iOS.

5.1 Struktura a návrh z vyšší perspektivy

Protože programátorská knihovna je produkt, který bude využívat po dlouhou dobu mnoho jiných programátorů, ne pouze já, považoval jsem za nezbytné na začátku své práce přemýšlet nad strukturou zdrojových kódů a jejich logickým uskupením. Zároveň požadavkem firmy Inmite s.r.o. byla jednoduchost použití pro ostatní programátory. Proto jsem se snažil komponenty knihovny navrhnout tak, aby splňovaly požadavek jednoduchosti a zároveň, aby byly co nejsnazší na pochopení. Bez nutnosti přikládat zdlouhavé manuály a návody na použití knihovny.

V této sekci uvedu základní stavební kameny knihovny a její strukturu z vyššího pohledu návrhových vzorů a logického seskupení zdrojových kódů.

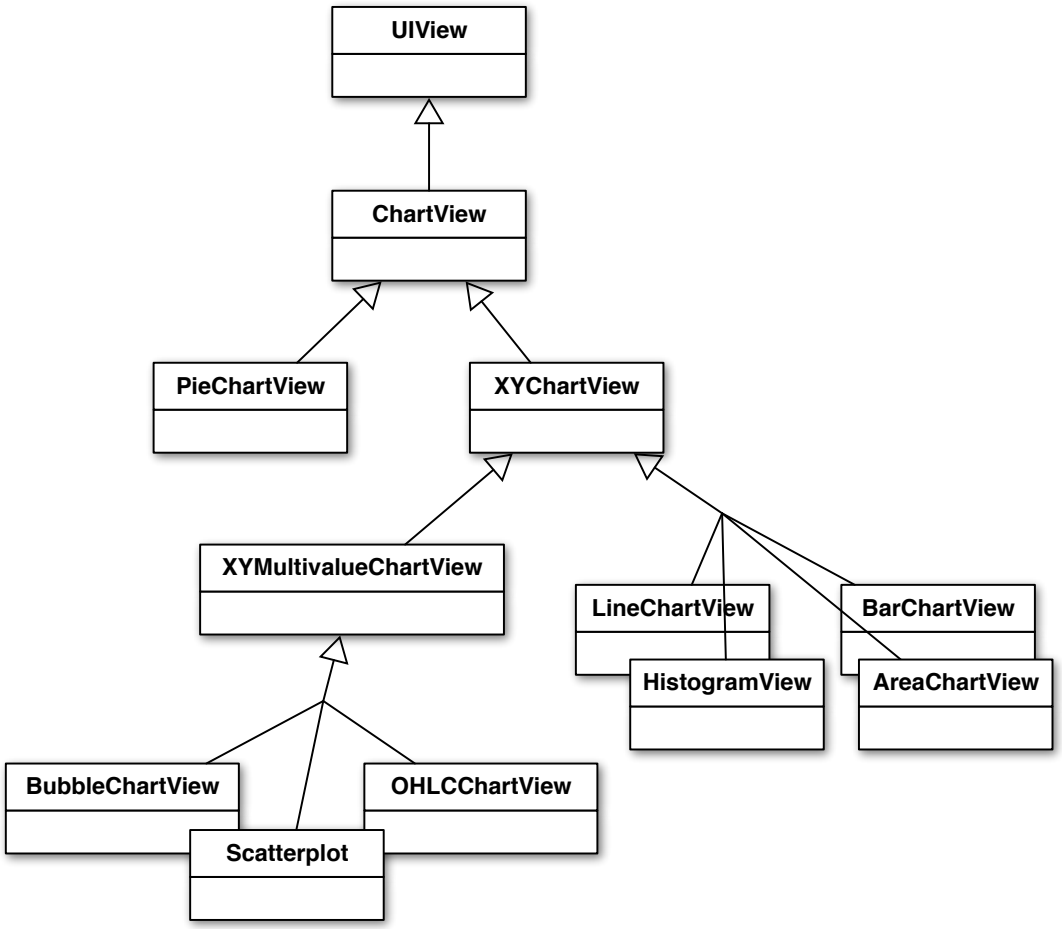
5.1.1 Použité návrhové vzory

5.1.1.1 Model–view–controller (MVC) První a velmi známý návrhový vzor, který mi „padl do oka“ byl MVC. Je to velmi rozšířený vzor mezi knihovnami a programovacími jazyky a samotný framework Cocoa Touch ho využívá velmi často. Původně jsem uvažoval, že každý graf, včetně titulku, legendy a ostatních náležitostí, bude v základní podobě vystupovat jako view controller a bude mít svůj vlastní, přesně definovaný pohled (view). Programátorovi, využívajícímu knihovnu, bude stačit jen definovat model, pomocí něhož se graf dozví, jaké data má zobrazit. Po delším přemýšlení jsem ale dospěl k názoru, že toto řešení je příliš omezující a neumožňuje rozsáhlejší možnosti rozšiřování tříd frameworku pro konkrétní použití.

Zvolil jsem tedy strategii, kdy je graf čistě jen pohled (view) a je možné jej napojit na zdroj dat. Samotný graf neobsahuje legendu či titulek. Tu je možné přiložit zvlášť. Ponechal jsem však možnost spojit graf, legendu a další části a vytvořit si view controller, který bude samostatný a nezávislý. Aktuální řešení je více variabilní z důvodu, že graf jako pohled lze vložit do jiných view controllerů a používat jej jako kterýkoliv jiný pohled. Na velmi podobném principu funguje také jedna z nejvíce používanějších komponent Cocoa Touch frameworku – `UITableView`, ze které jsem si pro návrh vzal hodně inspirace.

Hierarchická struktura pohledů a grafů je přiblížena na obrázku 4.

Všechny grafy jsou potomky standardní komponenty `UIView`, díky které je možné je vložit do view controlleru a obsluhovat je jako klasický pohled. Grafy, které jsou si podobné v tom smyslu, že jsou zobrazeny mezi dvěma osami hodnot x a y, navíc dědí



Obrázek 4: Třídní diagram hierarchie tříd grafů

z třídy `XYChartView`, která zajišťuje všechny potřebné výpočty, vykresluje osu, datovou mřížku atp., ale neumí vykreslit samotná data grafu. Konkrétní grafy jsou nadstavba nad touto třídou, která už spočítané hodnoty jen zobrazí buď jako čáru spojující jednotlivé body, jako sloupce vedené od osy X nebo jiným potřebným způsobem. Ale už se nemusí zajímat o vykreslení os, protože to je plně v režii mateřské třídy.

5.1.1.2 Data předávaná skrze objekt Data Source Možnosti napojení dat ke grafu už jsem probíral v dřívější části tohoto textu. Realizace je uskutečněna podle návrhového vzoru delegát, typického pro Objective-C a framework Cocoa Touch. Každý graf definuje delegáta určujícího přesně typ dat potřebných pro konkrétní graf.

Podobná dědičnost jako u grafů se dá vytvářet i u jejich delegátů. Třída `XYChartView` má svého delegáta `XYChartDataSource`, který požaduje data nutná pro zobrazení jakéhokoliv grafu mezi osami x a y. Jaká tyto data jsou naznačují metody delegáta, které jsou zobrazeny v následujícím výpise:

```
@protocol IMTXYChartDataSource <NSObject>
@required

– (NSUInteger)xyChartNumberOfPlots:(IMTXYChartView *)xyChart;
– (NSUInteger)xyChartNumberOfTicksOnAxisX:(IMTXYChartView *)xyChart;
– (NSString *)xyChart:(IMTXYChartView *)xyChart labelForAxisXTickAtIndex:(NSUInteger)
    tickIndex;
– (NSNumber *)xyChart:(IMTXYChartView *)xyChart valueForPointAtIndexPath:(NSIndexPath *)
    chartPath;

@end
```

Výpis 3: Definice protokolu `IMTXYChartDataSource`

Z metod Data Source vidíme, že každý graf, používající osy x a y potřebuje vědět, kolik křivek má zobrazit, kolik díků je na ose X a jaké jsou popisky k těmto dílkům. V poslední metodě se Data Source dotáže na konkrétní y hodnotu pro konkrétní křivku na konkrétním dílku osy x.

Pro vyjádření pozice v grafu (který dílek na ose X pro kterou čáru v grafu) se používá třída `NSIndexPath` rozšířená tak, aby obsahovala property `plot` označující pořadové číslo křivky a property `axisXTick` označující pořadí dílku na ose X.

Třídní hierarchie Data Source protokolů jednotlivých grafů je identická jako hierarchie samotných grafů, proto ji zde nebudu uvádět.

5.1.1.3 Zachycení dotykových událostí a reakce grafů Důkazem toho, že v průběhu implementace se velmi často mění struktura tříd a původních návrhů, je implementace zachycování dotykových gest v grafech. Původně byl můj plán využít stejného návrhového vzoru použitého pro Data Source. Každý graf by obsahoval svoji vlastní implementaci pro zachycování dotykových událostí a poskytoval by další property – Delegáta – oznamujícího, která událost byla právě vyvolána. To je také způsob, jakým řeší reakci na dotyková gesta třída `UITableView`.

Záhy jsem ale zjistil, že více grafů potřebuje reagovat na stejné dotykové události stejným způsobem a tím pádem by můj původní návrh způsobil kopírování stejného kódu do různých tříd. Proto jsem se pokusil vymyslet jiný způsob, který mi umožní znovu použití již existujícího kódu.

Nejprve jsem uvažoval o použití návrhového vzoru Dekorátor, ale nakonec jsem tento záměr zavrhl, protože nebyl dostačující. Mé výsledné řešení využívá objektů implementujících protokol `XYChartTouchBehavior`. Jak název protokolu napovídá, definuje dotykové chování použitelné jen pro grafy mezi osami x, y. Tento protokol obsahuje metodu, která chování grafu inicializuje a nastaví tak, aby bylo schopno spolupracovat s konkrétním grafem a dále obsahuje delegáta, který zajišťuje oznámení dotykových událostí způsobem, aby byly zachytitelné programátorem a ten je mohl dále zpracovat ve své aplikaci.

Třída `XYChartView` pak uchovává pole objektů implementujících tento protokol a zabezpečuje spolupráci grafu s těmito objekty. Tento přístup umožňuje přidání dalšího typu reakce na dotyková gesta pouze vytvořením nového objektu, implementujícího protokol `XYChartTouchBehavior`, a přiřazením objektu do příslušného pole v grafu. Graf obsluhuje všechny tyto objekty, takže přiřazením více objektů je možné docílit několika různých reakcí na různá dotyková gesta. Konkrétní způsob implementace tohoto vzoru upřesním v další části tohoto textu.

5.1.2 Oddělení kódu podle funkčnosti

Při psaní obecné knihovny je důležité nejen vytvořit dobrý návrh tříd a vazeb mezi nimi, ale také udržet přehlednou a oddělenou strukturu podle funkcionality a typu úlohy, které třídy vykonávají.

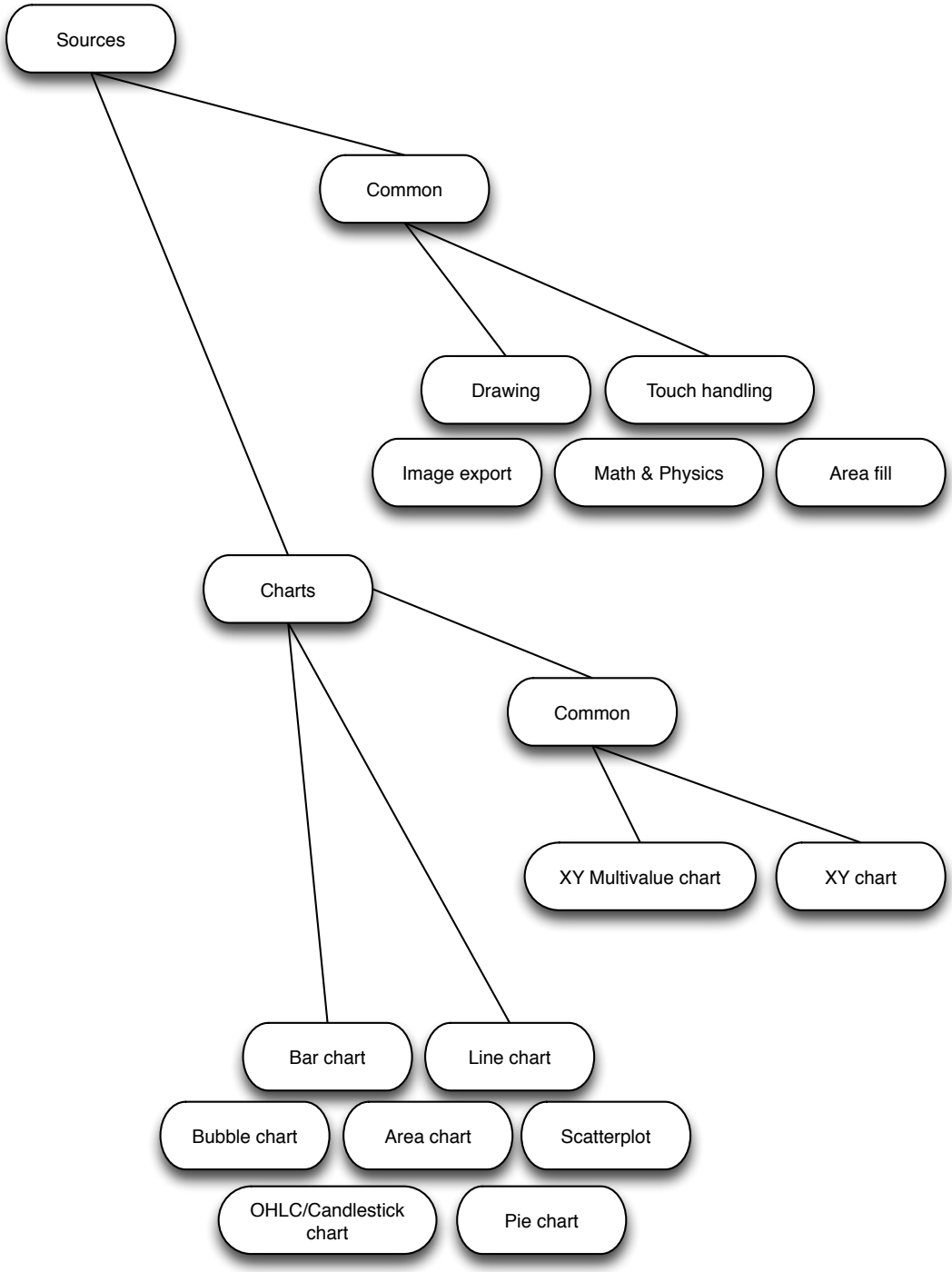
Já jsem se při psaní knihovny snažil o maximální oddělení logicky spolu souvisejícího kódu. Proto například třídy grafů neobsahují žádný kreslící kód. Využívají jen jiné prostředníky, kteří se o kreslení postarají. Stejně tak při zachycování a zpracovávání dotykových událostí se animace grafu dějí na základě matematických a fyzikálních výpočtů, ale grafy ani dotyková chování nedokáží vyhodnocovat fyzikální výpočty. Jen využívají třídy, které výpočty zabezpečují.

Struktura rozdělení kódu do kategorií je patrná z následujícího obrázku 5.

V následujících podkapitolách tyto kategorie detailněji popíšu.

5.1.2.1 Matematické a fyzikální výpočty Ačkoliv to na první pohled není zřejmé, některé grafy potřebují notnou dávku matematiky a fyziky, aby docílily přirozených animací. Je to zejména koláčový graf, který při svém otáčení zjišťuje úhlovou rychlost, po ukončení tažení prstem pokračuje otáčení pohybem rovnoměrně zpomaleným, a před zastavením vytváří efekt magnetického přitahování do středu výseče, čehož je docíleno tlumeným kmitavým pohybem.

Pro počítání matematických a fyzikálních výpočtů jsem zvolil nejjednodušší možnou implementaci. Jednotlivé funkce jsou implementovány jako statické metody tříd. Toto řešení jsem zvolil z toho důvodu, že potřebuji využívat velmi rozdílné matematické a fyzikální výpočty a nemělo by velký význam vymýšlet složitou hierarchii tříd a protokolů



Obrázek 5: Struktura rozdělení kódu podle oblastí působnosti

obsluhujících matematiku. Výpočty jsem jen podle jejich vlastností rozdělil do tříd `Math`, `Math2D` a `Physics`. Samotný výpočet je tedy realizován zavoláním statické metody příslušné třídy a předáním potřebných argumentů této metodě.

Ukázka zjišťování výsledku fyzikálního výpočtu pro otáčení grafu:

```
CGFloat y1 = [IMTPhysics
amplitudeForDampedOscillationWithMaximumAmplitude:_oscilationMaximumAmplitude
                                    radialVelocity : _oscilationVelocity
                                      initialPhase :_oscilationPhase
                                              time:_oscillationTime
                                           damping:damping];
```

Výpis 4: Volání fyzikální funkce kmitavého pohybu

Díky zvolené úrovni oddělení kódu je také mnohem jednodušší provádění unit testů. Všechny matematické a fyzikální výpočty jsou otestovány unit testy, aby se zajistila jejich bezchybná funkčnost.

5.1.2.2 Vykreslování Veškerý kód, který obstarává vykreslování, je soustředěn do tříd implementujících jednoduchý protokol `Drawable`. Tento protokol obsahuje jedinou metodu `- (void)drawInContext:(CGContextRef)context`. Každá třída implementující protokol `Drawable` musí prostřednictvím této metody zajistit, že se dokáže správně vykreslit v grafickém kontextu. Grafy obsažené v knihovně potřebují vykreslovat čáry, kruhové výseče či text. Pro každý z těchto vykreslitelných objektů je definována třída obsahující property a metody pro vytvoření správného objektu. A protože každý z objektů implementuje metodu `drawInContex:`, je možné jej snadno vykreslit na displej zařízení. Navíc díky zabalení vykreslování do tříd je možné jejich instance ukládat a znovu je používat bez nutnosti jejich opětovného inicializování.

5.1.2.3 Výplň grafových oblastí Další logicky oddělitelnou částí kódu je tvorba výplně pro grafy. Identickou výplň totiž může používat graf sloupcový pro vnitřek sloupců, graf spojnicový pro výplň oblasti pod křivkou dat a jiné grafy. Proto jsem tento kód oddělil do zvláštních tříd a umožnil tak jeho znovupoužití kdekoliv v aplikaci.

Klíčovým pro definici výplně je protokol `AreaFill`:

```
@protocol IMTAreaFill <NSObject>
@required

@property (nonatomic, assign) CGFloat strokeWidth;
@property (nonatomic, retain) UIColor *strokeColor;

- (void)drawInRect:(CGRect)rect;
@end
```

Výpis 5: Definice protokolu `IMTAreaFill`

Metoda `drawInRect:` má za úkol vykreslit výplň ve vymezené oblasti. Dále protokol obsahuje property, umožňující specifikovat barvu a tloušťku ohraničení výplně. Každý typ výplně, ať je to jednolitá barva, barevný přechod nebo obrázková výplň, musí

implementovat tento protokol. Díky této obecnosti graf nepotřebuje znát typ své výplně. Pracuje jen s obecným protokolem a nechává samotné vykreslení výplně na konkrétním objektu.

5.1.2.4 Implementace grafů Velmi přirozeným rozhodnutím bylo sdružit dohromady třídy související s jednotlivými grafy. Třídy tvořící implementaci jednoho konkrétního grafu začínají vždy stejným prefixem, vyjadřujícím typ grafu, a poté následuje určení role třídy. Tedy například u koláčového grafu mají třídy názvy `PieChartView`, `PieChartDataSource`, `PieChartDelegate`, `PieChartSlice` apod.

5.1.2.5 Privátní a veřejné třídy a metody, kategorie Z hlediska bezpečnosti kódu je také vhodné uvažovat nad tím, které třídy budou veřejné a viditelné zvenčí a které budou sloužit jen pro interní potřebu knihovny a neměly by být přístupné programátorům používajícím knihovnu.

Samotný framework Cocoa Touch, obsahuje velké množství privátních tříd. Nic však není stoprocentní. Existují způsoby jak privátní třídy odhalit a jak je používat. Společnost Apple se však snaží všemožnými způsoby volání soukromých tříd zabránit a pokud zjistí tuto skutečnost, většinou to způsobí zamítnutí schválení aplikace do AppStore, jediného místa, kde je možné aplikace prodávat.

Já ve své knihovně také používám hrstku soukromých tříd, u kterých nechci, aby byly viditelné zvenčí. Pokud se někde privátní třídy vyskytnou, jsou umístěny ve zvláštní složce s názvem *private*. Jsou to většinou jednoúčelové implementační úseky, vztahující se ke konkrétnímu problému nebo grafu.

Avšak i veřejné třídy obsahují podstatnou část metod, které jsou privátní. Snažím se všechny nepotřebné záležitosti schovávat před okem programátora, aby přístupné rozhraní zůstalo co nejjednodušší a zároveň umožňovalo budoucí rozšíření. Ve svém kódu využívám privátní metody velmi často. Odhadem více než polovina metod, které třídy mají, jsou privátní. Vychází to také ze skutečnosti, že je vhodnější psát kratší metody zajišťující jednu konkrétní činnost a ty pak používat vícekrát. Přesně takovéto metody jsou adepty na to, stát se privátními.

5.1.2.6 Zvyky pro psaní privátních metod Objective-C podporuje tzv. rozšiřování tříd (class extensions). Zápis rozšíření je podobný definici rozhraní třídy, ale za názvem třídy se ještě přidávají prázdné závorky `()`, a zároveň se už nespecifikují mateřské třídy, ze kterých třída dědí.

Ukázka rozšíření třídy `PieChartView`:

```
@interface IMTPieChartView () {
@private
    NSMutableArray *_dataValues;
    NSMutableArray *_colors;
    NSInteger _numberOfSlices;
    NSUInteger _selectedSliceIndex;
    struct {
        unsigned int dsNumberOfSlicesExists:1;
```

```
        unsigned int dsSliceAtIndexExists:1;
    } _pieViewFlags;
}
@property (nonatomic, retain) NSMutableArray *dataValues;
@property (nonatomic, assign) NSInteger numberOfSlices;
@property (nonatomic, assign) NSUInteger selectedSliceIndex;

- (void)_initializeDataSourceFlags;
- (void)_initializeDelegateFlags;
- (void)_prepareForDrawing;
@end
```

Výpis 6: Rozšíření třídy `IMTPieChartView`

Dále se s rozšířením pracuje jako s klasickou definicí třídy – mohou se přidávat privátní proměnné, property či metody. Nepsaným pravidlem také je, že všechny privátní metody začínají svůj název podtržítkem.

Rozšíření se definuje v souboru s implementací třídy (`.m`) v horní části souboru. Tím se zajistí, že uvnitř implementace jsou viditelné všechny metody, i ty privátní, ale navenek lze použít jen ty, které jsou definované ve veřejném hlavičkovém souboru.

5.1.2.7 Kategorie – rozšíření třídy bez nutnosti z ní dědit Na velmi podobném principu, jako jsou třídní rozšíření, fungují také kategorie. Ve skutečnosti je třídní rozšíření speciálním typem kategorie, která nemá žádný název.

Kategorie je tedy další způsob, jak třídu rozšířit o nové metody. Tento způsob definuje veřejně použitelné metody všude, kde vložíme hlavičkový soubor kategorie. Velká síla kategorií je v tom, že s jejich pomocí je možné přidávat metody do existujících tříd, jejichž implementaci nemůžeme nebo nechceme měnit a dokonce i do tříd, jejichž implementaci máme dostupnou jen ve zkompilované binární podobě. To vše bez nutnosti použití dědičnosti. Pomocí kategorií dokážeme jakékoliv třídě fakticky přidat nové metody. Tato vlastnost je typická pro jazyk Objective-C a nevyskytuje se v jiných známých jazycích, jako je Java nebo C++.

Dříve jsem zmínil, že třídní rozšíření je kategorie beze jména, z toho plyne, že každá kategorie musí mít své jméno. Druhým rozdílem mezi kategoriemi a rozšířeními tříd je, že kategorie nedovolují definovat privátní proměnné, pouze property nebo nové metody. Jinak je syntaxe rozhraní kategorie identická se syntaxí třídního rozšíření. Pro implementaci platí, že za název třídy se navíc v závorkách připisuje název kategorie.

Pro ukázku přikládám implementaci jednoduché kategorie ke třídě `NSIndexPath`.

Definice:

```
@interface NSIndexPath (IMTXYChartView)
+ (NSIndexPath *)indexPathForAxisXTick:(NSInteger)axisXTick inPlot:(NSInteger)plot;
@property(nonatomic,readonly) NSInteger axisXTick;
@property(nonatomic,readonly) NSInteger plot;
@end
```

Výpis 7: Definice kategorie `NSIndexPath`

A implementace:

```
@implementation NSIndexPath (IMTXYChartView)
+ (NSIndexPath *)indexPathForAxisXTick:(NSInteger)axisXTick inPlot:(NSInteger)plot {
    return [self indexPathForRow:axisXTick inSection:plot];
}
- (NSInteger)axisXTick {
    return self.row;
}
-(NSInteger)plot {
    return self.section;
}
@end
```

Výpis 8: Implementace kategorie `NSIndexPath`

5.2 Zajímavé ukázky z implementace knihovny

Při tvorbě knihovny jsem se setkal s některými implementačními úseky, které bych na tomto místě rád zmínil. Některé z nich přibližují myšlenky, které jsem naznačil dříve v textu, jiné popisují situace, které nejsou na první pohled patrné a hledání jejich správného řešení vyžadovalo větší míru experimentování a testování různých přístupů v implementaci.

5.2.0.8 Zachycování dotykových událostí Jak jsem již dříve zmínil, v průběhu implementace zachycování dotykových událostí jsem zjistil, že potřebuji přepracovat své původní plány a navrhl jsem proto rozhraní `XYChartTouchBehavior`, které jsem popsal v dřívější části tohoto textu. Nyní detailně rozeberu, jak zachycování gest pomocí dotykových chování funguje. Předtím však ukáži, jak se obecně řeší zachycení dotyků na iOS.

Pro zachycení dotykových událostí je velmi výhodné použít podtřídy `UIGestureRecognizer`. Protože tyto třídy definované frameworkem Cocoa Touch umožňují velmi jednoduché zachycení konkrétních gest – ťuknutí (*tap*), tažení prstem (*pan*), otáčení (*rotate*), roztahování (*pinch*), a mnoho dalších.

Práce s gesture recognizery je následující:

1. **vytvoření gesture recognizeru**
   Při vytváření gesture recognizeru specifikujeme `target` a `action` neboli objekt a jeho metodu, která se na něj má zavolat při sledování průběhu gesta.

   ```
   _panGesture = [[UIPanGestureRecognizer alloc] initWithTarget:self action:@selector(
       panChartArea:)];
   ```

   Výpis 9: Inicializace gesture recognizeru

2. **přiřazení recognizeru k pohledu**
   Pohledu, ve kterém se gesto má zachytávat, se přiřadí gesture recognizer pomocí metody `addGestureRecognizer:`

```
[self addGestureRecognizer:_panGesture];
```

Výpis 10: Přiřazení recognizeru k pohledu

3. **sledování průběhu gesta**
   Průběh gesta sledujeme právě metodou, kterou jsme specifikovali při inicializaci recognizeru.

   Tato metoda musí přebírat jeden parametr, kterým je samotný gesture recognizer. Recognizer ji zavolá ve chvíli, kdy se nějak změní stav gesta a do parametru předá sám sebe. Uvnitř metody, prostřednictvím objektu recognizeru, zjistíme souřadnice doteku v pohledu, stav gesta (započetí, průběh, ukončení,..) a jiné vlastnosti. Některé vlastnosti jsou specifické pro konkrétní gesta, například gesto pro tažení prstem poskytuje informaci o rychlosti pohybu prstu, gesto pro rotaci zná úhel aktuálního otočení, apod. Zde vidíme sílu použití recognizerů. Programátor tyto údaje nemusí sám počítat, ale recognizer je spočítá za něj.

Výpis nejčastěji zachycovaných stavů recognizeru:

- `UIGestureRecognizerStatePossible` – z obdržených dotyků ještě nebylo konkrétní gesto rozpoznáno, ale je možné, že se tomu tak stane po zachycení více dodatečných dotyků
- `UIGestureRecognizerStateBegan` – z dotyků bylo rozpoznáno, že se jedná o požadované gesto
- `UIGestureRecognizerStateChanged` – dotyky déle trvajícího gesta pokračují
- `UIGestureRecognizerStateEnded` – z dotyků bylo rozpoznáno správné ukončení gesta
- `UIGestureRecognizerStateCancelled` – recognizer obdržel dotyky, které zapřičinily ukončení gesta (například při rozpoznání jiného gesta, které pohled souběžně zachycuje)
- `UIGestureRecognizerStateFailed` – recognizer obdržel sekvenci několika dotyků, kterou není schopen rozeznat jako součást žádného gesta

Nyní jsme si ukázali, jak používat gesture recognizery. To je však jen jedna část samotného reagování na dotyková gesta. Poté, co gesto v grafu zachytím, chci na něj reagovat a reflektovat změny v pohledu. To se děje v metodě zachycující stav gesta. Avšak je nutno mít na paměti fakt, že v této metodě nemůžeme vykreslovat uživatelské rozhraní pomocí funkcí CoreGraphics, protože jak už jsem zmínil dříve, vykreslování je možné jen v určité správné chvíli, což však není chvíle, kdy je zachycena událost. Až tato chvíle přijde, bude zavolána metoda `drawRect:`. Proto se při zachycení dotyku pouze poznačí údaje o dotyku a přepočítají se hodnoty potřebné pro dodatečné kreslení a na sledovaný pohled se zavolá metoda `setNeedDisplay`. Ta způsobí, že pohled se překreslí, jak nejdříve to

bude možné. V metodě `drawRect:` se pak podle vypočítaných hodnot vykreslí to, co je potřeba.

Přesně na tomto principu fungují v grafové knihovně třídy implementující protokol `XYChartTouchBehavior`. Navíc to jsou třídy, které dědí z `UIView`. Přiřazením dotykového chování do grafu se objekt chování vloží jako sub-pohled ke grafu a nastaví sám sobě dotykový recognizer. V hierarchii pohledu je objekt dotykového chování umístěn nad grafem a celý ho překrývá. Při zachycení dotyku překresluje sám sebe a není tedy nutné znova vykreslovat celý graf, což je efektivní z hlediska časové náročnosti. Navíc toto uspořádání umožňuje široké možnosti zvýraznění grafu, aniž by bylo ovlivněno samotné zobrazení dat. Objekt dotykového chování si také drží instanci grafu, ve kterém zachycuje události, takže má možnost i ovlivnit tento graf, změnit ho, případně ho překreslit.

Ukázkovou implementaci takového objektu si lze prohlédnout na přiloženém CD v souboru `/src/lib/IMTCharts/IMTCharts/IMTXYChartTouchPanBehavior.m`.

5.2.1 Přepočítání hodnot grafu těsně před jeho vykreslením

Požadované chování grafů je takové, že programátor může kdykoliv nastavit nějakou property a graf by měl toto nastavení správně reflektovat. To však není úplně triviální problém, vzhledem k tomu, že graf ideálně potřebuje vypočítat hodnoty dat, která má zobrazit, dříve, než během jejich vykreslování. Je tedy potřeba najít způsob, jak zajistit, že hodnoty budou přepočítány před samotným vykreslováním, ale ne příliš brzo, aby dokázaly reflektovat změnu hodnot grafu, kterou programátor provedl.

Počítání těchto hodnot hned při inicializaci grafu by bylo nevhodným řešením, protože po inicializaci bývá graf mnohokrát změněn, například je připojen Data Source. Jako rozumné řešení se zdá vypočítat hodnoty grafu po přiřazení Data Source objektu, ale tím by se už nereflektovaly změny, které se s grafem provedou poté, co už byl Data Source přiřazen. A to opět nesplňuje požadavek reflektování všech změn, provedených kdykoliv.

Při důkladném prozkoumání rozhraní třídy `UIView`, ze které všechny grafy dědí, jsem zjistil, že `UIView` neposkytuje žádný způsob, jak získat notifikaci o tom, že se právě chystá být překreslen. Jediné místo, které lze s jistotou použít pro rozpoznání, že graf bude zobrazen, je metoda `drawRect:`, tedy metoda, která zajišťuje samotné vykreslení. To si ale mírně protiřečí s tím, co jsem zmínil dříve – při vykreslování grafu by se už nemělo nic počítat. Proto jsem se snažil co nejvíce počítání optimalizovat a vykonávat ho jen tehdy, když je skutečně potřeba, když už ho tedy budu muset dělat v metodě pro vykreslování grafu.

```
- (void)drawRect:(CGRect)rect {

    if (!_dataIsValid) {
        [self prepareForDrawing];
    }

    if (self.showAxisX)
        [self drawAxisXInRect:self.axisXFrame];
```

```
    [self drawChartAreaInRect:self.chartAreaFrame];

    if (self.showAxisY)
        [self drawAxisYInRect:self.axisYFrame];
}
```

Výpis 11: Implementace metody `drawRect:` ve třídě `XYChartView`

Zavedl jsem privátní booleovskou proměnnou `_dataIsValid`, která udržuje informaci o tom, jestli jsou data grafu aktuální. Data grafu se přepočítávají jen tehdy, pokud validní nejsou. Naopak, pokud dojde k požadavku vykreslení grafu, jehož data jsou stále validní, není potřeba je znovu vyhodnocovat a okrádat se tak o výpočetní čas.

5.2.2 Rotace koláčového grafu pomocí časovačů a fyzikálních funkcí

Implementace otáčení koláčového grafu nebyla vůbec jednoduchá. Nejnáročnější bylo zejména setrvačné otáčení a také efekt magnetického přitahování středu výseče při ukončování rotace. Nejdříve jsem se pokoušel tyto animace implementovat pomocí třídy `CAAnimation` a standardních časovacích funkcí z Cocoa Touch. Ale výsledek nebyl ideální a nevypadal přirozeně. Dospěl jsem tedy k názoru, že pro skutečně realistický výsledek musím do hry zapojit skutečnou fyziku, a proto bylo potřeba vytvořit fyzikální funkce, které dokáží simulovat reálné otáčení kotouče.

Oba druhy otáčení, setrvačné i magnetické přitahování do středu, fungují na stejném principu. Je vytvořen časovač, který několikrát, podle frekvence oken za sekundu, volá metodu, ve které se spočítá posunutí za daný časový okamžik a vyvolá se překreslení grafu. Pro zjištění posunu se využívají fyzikální funkce pro rovnoměrně zpomalený pohyb (1) a pro tlumené kmitání (2).

$$s = v \cdot t - 1/2 \cdot a \cdot t^2 \quad (1)$$

$$y = e^{-d \cdot t} \cdot Y \cdot sin(\omega \cdot t + \phi) \quad (2)$$

Před samotným započetím animace je nutné spočítat počáteční hodnoty nezbytné pro fyzikální výpočet. Pro kmitání kolem středu výseče jsou to hodnoty kmitavého pohybu – počáteční fáze, maximální amplituda, počáteční úhlová rychlost a poloměr rotace. Navíc, protože každá kruhová výseč je jinak veliká a pro správný efekt musí kmitavý pohyb probíhat v rozmezí odchylky $<\frac{\pi}{2}; -\frac{\pi}{2}>$, probíhá na začátku mapování úhlové vzdálenosti výseče na tento interval, ve kterém se kmitání počítá.

```
_oscillationTransformationRatio = M_PI_2 / maxAngle;

_oscilationPhase = [self angleToMiddleOfSliceWithIndex:sliceIndex] *
    _oscillationTransformationRatio;
_oscilationMaximumAmplitude = _wheelView.wheelRadius * _oscillationTransformationRatio;
_oscilationRadius = _oscilationMaximumAmplitude;
_oscillationTime = 0;
```

```
CGFloat zeroVelocity = 3.0f;
_oscilationVelocity  = (  initialVelocity   + zeroVelocity) * _oscillationTransformationRatio;

_oscilationTimer = [NSTimer scheduledTimerWithTimeInterval:1.0f/kRotationFPS
                                                    target:self
                                                  selector:@selector(oscilationRotationFrame:)
                                                  userInfo:nil
                                                   repeats:YES];
```

Výpis 12: Inicializace hodnot pro animaci magnetického přitahování do středu výseče koláčového grafu

Při určování posunu v každém okně animace se tato vzdálenost opět přepočítává na skutečnou vzdálenost ve výseči. O tuto vzdálenost se pak posune koláč grafu a pokračuje se dalším krokem animace.

```
// vypocitani okamzite odchylky v case t0 a t1 pomoci promennych _oscilationMaximumAmplitude,
// _oscilationVelocity , _oscilationPhase, _oscillationTime a~fyzikalnich funkci
// y0 = ...
// y1 = ...

CGFloat fi0 = asinf(y0 / _oscilationRadius) / _oscillationTransformationRatio;
CGFloat fi1 = asinf(y1 / _oscilationRadius) / _oscillationTransformationRatio;

CGFloat angleShift = fi0 - fi1;
_rotationImmediateVelocity = angleShift / timeStep;

_wheelView.transform = CGAffineTransformRotate(_wheelView.transform, -angleShift);
[self didRotateWithAngle:angleShift];
```

Výpis 13: Vypočítání okamžité odchylky při oscilaci koláčového grafu

5.2.3 Animované načítání grafů

Jedním z mých požadavků na knihovnu bylo animované načítání grafů. Jednak proto, že existující knihovny animace téměř nepodporují, a také hlavně proto, že animace jsou standardním prvkem iOS aplikací, na které jsou uživatelé zvyklí a očekávají je. Právě pěkné animace jsou jedním z tzv. „wow efektů“, jež může vývojář do aplikace zakomponovat a zaujmout tak svého zákazníka. Pokud si uživatel hraje s nějakým prvkem grafického rozhraní jen, protože se mu líbí animace, je to velký úspěch pro programátora. O podobný úspěch jsem se snažil i já při návrhu animací v grafech. Samozřejmě všeho musí být s mírou a animace by měly být decentní a rychlé. Standardní doba trvání animace se pohybuje okolo $1/3$ sekundy.

Všechny grafy zobrazené mezi osami x a y využívají jednotný princip animací. Tento princip ukážu na příkladu animace histogramu. Pro připomenutí, animace histogramu má 3 části:

1. vyrůstání sloupců grafů od osy x směrem nahoru
2. zobrazování křivky kumulované četnosti zleva doprava

3. zobrazení číselných hodnot křivky v závěru animace

Animace začíná zavoláním metody `reloadDataAnimated:`, která ji inicializuje a spustí časovač aktualizující v pravidelném intervalu graf tak, aby zobrazoval jednotlivá okna animace:

```
- (void)reloadDataAnimated:(BOOL)animated {
    [super reloadDataAnimated:animated];

    if (animated) {
        _animationTimeStep = 0;
        _animationDistortion = CGSizeMake(1.0f, 0.0f);

        [NSTimer scheduledTimerWithTimeInterval:1.0f/kAnimationFPS target:self selector:
            @selector(animationStep:) userInfo:nil repeats:YES];
    }
}
```

Výpis 14: Implementace metody Histogramu `reloadDataAnimated:`

V každém kroku animace je zavolána metoda `animationStep:`, jejímž úkolem je upravit hodnotu proměnné `_animationDistortion` a vyvolat překreslení grafu pomocí metody `setNeedsDisplay`.

```
- (void)animationStep:(NSTimer *)timer {

    _animationDistortion = CGSizeMake(1, [self displacementsForDefaultExponentialAnimation][
        _animationTimeStep++]);
    [self setNeedsDisplay];

    if (_animationTimeStep >= kAnimationTimeSteps) {
        [timer invalidate];
        _animationDistortion = CGSizeMake(1.0f, 1.0f);
    }
}
```

Výpis 15: Krok animace v histogramu

Při vykreslování grafu je využito faktu, že proměnná `_animationDistortion` nabývá hodnot z intervalu $\langle 0; 1 \rangle$ v závislosti na aktuálním průběhu animace – na začátku je nulová, na konci má hodnotu 1.

V prvním kroku animace, nárůstu sloupců, se vynásobí konečná výška každého sloupce hodnotou `_animationDistortion`, tím se zajistí, že během animace sloupce porostou podle této hodnoty.

Dalším krokem je postupné zobrazování křivky kumulativní četnosti zleva doprava. Tohoto efektu je docíleno tak, že křivka se ve skutečnosti vykreslí celá, ale poté se na ni aplikuje omezující obdélník, tzv. clipping path (tento princip bude popsán níže v textu), který způsobí vykreslení jen části křivky. Podle hodnoty `_animationDistortion` se mění šířka obdélníku, od nulové po šířku celého grafu.

Konečně posledním krokem je uzavření animace zobrazením číselných hodnot popisujících křivku kumulativní četnosti. Toho docílíme jednoduchou podmínkou, říkající, že

text se vykreslí až tehdy, pokud se hodnota `_animationDistortion` blíží číslu 1, což naznačuje ukončení animace.

Všechny tyto kroky jsou zpracovávány v metodě `drawChartAreaInRect:` sloužící k vykreslení grafu pomocí grafického kontextu. Zdrojový kód této metody je zahrnut sekci A jako příloha A.4.

5.2.4 Ukládání `UIView` do obrázku nebo PDF souboru

Díky nezávislosti objektu `UIView` na kontextu, ve kterém se vykresluje, je velmi jednoduché uložit pohled jako obrázek nebo PDF soubor. Stačí si jen místo kontextu kreslícího na displej vytvořit PDF kontext nebo obrázkový kontext a nechat do něj existující pohled znovu vykreslit pomocí metody `[CALayer renderInContext:]`. PDF kontext se vytváří pomocí volání funkce `CGPDFContextCreateWithURL`, jak lze vidět na následující ukázce kódu:

```
CGRect mediaBox = self.bounds;
CGContextRef ctx = CGPDFContextCreateWithURL((__bridge CFURLRef)[NSURL fileURLWithPath
    :path], &mediaBox, NULL);

CGPDFContextBeginPage(ctx, NULL);
CGContextScaleCTM(ctx, 1, -1);
CGContextTranslateCTM(ctx, 0, -mediaBox.size.height);

[self.layer renderInContext:ctx];

CGPDFContextEndPage(ctx);
CFRelease(ctx);
```

Výpis 16: Metoda kategorie `UIView` ukládající pohled jako PDF soubor

Pokud chceme pohled uložit do obrázku, je potřeba analogicky vytvořit kontext obrázkový, pomocí funkce `UIGraphicsBeginImageContextWithOptions`. Díky tomuto kontextu dokážeme z pohledu vytvořit objekt `UIImage`, se kterým můžeme dále libovolně nakládat. Například uložit ho do galerie obrázků v telefonu pomocí funkce `UIImageWriteToSavedPhotosAlbum`, odeslat ho jako přílohu emailem nebo ho uložit jako soubor do dokumentů aplikace a pracovat s obrázkem dále.

```
UIGraphicsBeginImageContextWithOptions(self.bounds.size, self.opaque, 0.0);
[self.layer renderInContext:UIGraphicsGetCurrentContext()];

UIImage* image = UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();
```

Výpis 17: Transformace objektu `UIView` na objekt `UIImage`

Další částí skládanky jsou funkce, které z instance třídy `UIImage` dokáží vytvořit `.jpg` nebo `.png` soubor. Jsou to funkce `UIImagePNGRepresentation` a `UIImageJPEGRepresentation`.

V knihovně jsou tyto funkce implementovány v samostatné kategorii třídy `UIView`. Díky tomuto přístupu je možné kterýkoliv pohled uložit jako obrázek či PDF soubor nebo jej převést na objekt `UIImage`.

Kompletní kód kategorie je k nahlédnutí v sekci A jako příloha A.5.

5.3 Grafický výstup pomocí CoreGraphics

Veškeré vykreslování grafů je zabezpečováno frameworkem CoreGraphics, který už byl nepatrně zmíněn v dřívější části tohoto textu. V této části framework popíši důkladněji s důrazem na jeho reálné použití při implementaci vykreslování objektů.

5.3.1 Základní princip vykreslování jednoduchých objektů

Základním stavebním kamenem je grafický kontext, který uchovává svůj aktuální grafický stav. V principu jde o to, změnit nějakým způsobem grafický stav kontextu a poté kontext vykreslit. Stav se mění pomocí funkcí, které začínají předponou `GCContext`. Například barvu výplně lze změnit funkcí `CGContextSetFillColor`, šířku vykreslené čáry lze nastavit funkcí `CGContextSetLineWidth` a spousty dalších. Některé objekty Cocoa Touch, například `UIColor`, či `UIBezierPath` mají své metody, které samy změní vlastnosti grafického kontextu. Například nastavení barvy výplně lze docílit také pomocí metody `[UIColor setFill]`.

Kontext, který můžeme vykreslit, získáme jako návratovou hodnotu funkce `UIGraphicsGetCurrentContext`.

```
CGContextRef context = UIGraphicsGetCurrentContext();

CGContextSetLineWidth(context, 2);
[[UIColor redColor] setStroke];
[[UIColor greenColor] setFill ];
CGContextMoveToPoint(context, 10, 10);
CGContextAddLineToPoint(context, 10, 100);
CGContextAddLineToPoint(context, 100, 100);
CGContextAddLineToPoint(context, 100, 10);
CGContextAddLineToPoint(context, 10, 10);
CGContextDrawPath(context, kCGPathFillStroke);
```

Výpis 18: Ukázka vykreslení zeleného čtverce s červeným okrajem

Z předchozí ukázky kódu vidíme, že kreslení probíhá na kanvasu určeném velikostí pohledu. Vždy je potřeba kontext umístit do určitého bodu a říci mu, jak má kreslit. V tomto případě jsme vytvořili cestu (path) pomocí posunutí kontextu do počátečního bodu a několikerého přidání úsečky k cestě použitím funkce `CGContextAddLineToPoint`. Informace o cestě se ukládají uvnitř kontextu, zavoláním funkce `CGContextDrawPath` se celý obsah kontextu vykreslí.

Samozřejmě cestu nemusíme vytvářet jen za pomoci úseček. Můžeme do ní přidávat také části kružnic, elipsy, obdélníky nebo libovolné křivky (Beziérovy, ale i jiné typy). Pro ilustraci, graf `LineChartView` automaticky vytváří křivku podle aktuálních dat zobrazených v grafu a tuto křivku (neboli cestu) zobrazuje. K tomu se využívá metody

`[UIBezierPath stroke]`, která upraví kontext tak, aby cestu vykreslil jako jeden tah štětcem.

Zjednodušená ukázka vykreslení spojnicového grafu:

```
UIBezierPath *path = [self closingPathForPlotWithIndex:i];
//
// .. uprava cesty do pozadovaneho stavu (barva, tloustka,..)
//
[color setStroke];
[path stroke];
CGContextDrawPath(context, kCGPathStroke);
```

Výpis 19: Ukázka vykreslení cesty pomocí CoreGraphics

Často je také potřeba vykreslit text. Toho lze velmi snadno docílit díky metodě `[NSString drawInRect:withFont:]`, která text přímo zobrazí na kanvas pomocí aktuálního grafického kontextu. Text bude vykreslen v daném obdélníku daným fontem. Před tím, než text budeme kreslit, většinou chceme vědět, jak velký obdélník vlastně bude zabírat. Pro zjištění této velikosti můžeme použít metodu `[NSString sizeWithFont:]` vracející velikost textu zobrazeného daným fontem.

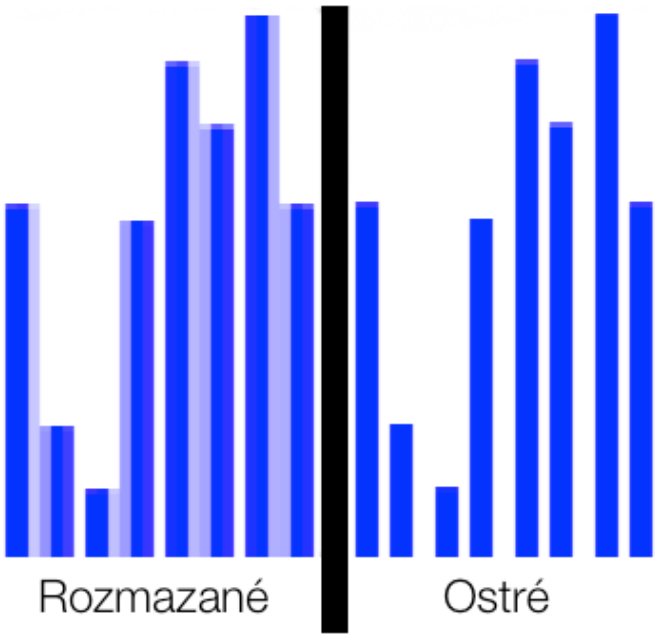
```
UIFont *font = [UIFont systemFontOfSize:9.0f];
NSString *label = [ self .dataSource xyChart:self labelForAxisXTickAtIndex:i];
CGSize labelSize = [label sizeWithFont:font];
CGRect labelFrame = CGRectMake(0, 0, labelSize.width, labelSize.height);

[label drawInRect:labelFrame withFont:font];
```

Výpis 20: Ukázka vykreslení textu

5.3.2 Pixel perfect vykreslování

Při kreslení tenkých čar a některých detailů se může stát, že výsledek bude zobrazen rozmazaně. Souvisí to s hardwarovým omezením kreslení. CoreGraphics používá pro souřadnice bodu hodnoty typu `float`, takže je možné například vykreslit čáru širokou 2,5 px. Ale samozřejmě fyzický pixel nelze rozdělit na polovinu. Takže čára o tloušťce 2,5 px se ve skutečnosti vykreslí na tři pixely a poslední pixel bude vybarven světlejší barvou, což simuluje použití jen poloviny pixelu. Díky tomuto jevu může být vykreslování tenkých čar rozmazané, pokud si programátor nedá pozor na tuto skutečnost a nepřizpůsobí tomu souřadnice počátku kreslení.

Při vytváření čáry vezme CoreGraphics aktuální pozici kontextu (lze nastavit například pomocí funkce `CGContextMoveToPoint`) a tato pozice se stane středem čáry. Kolem tohoto středu se čára kreslí vždy na každé straně polovinou své tloušťky. Pokud máme například čáru tlustou 2 px a začínáme kreslit na pozici $[10, 0]$, bude obarven 10. a 11. pixel od počátku. Nyní, když si představíme, že čára je široká pouze 1 pixel, nastává problém, protože tehdy by se správně měla zabarvit pravá polovina 10. pixelu a levá polovina 11. pixelu. To však není fyzicky možné, proto se zabarví oba pixely, ale světlejší barvou.



Obrázek 6: Demonstrace grafického výstupu optimalizovaného pro pozice pixelů

Řešením problému rozmazaných pixelů je sledovat tloušťku čáry a počátek vykreslování. Při liché tloušťce čáry posuneme počátek o 0,5 px některým směrem a díky tomu se výsledek *strefí* přesně do pozic pixelů. V předchozím případě bychom například zvolili tloušťku čáry 1 px a počáteční pozici $[10,5;0]$, nyní by se zabarvil pouze jeden pixel, 10. v pořadí.

Na obrázku 6 je porovnání výstupu sloupcového grafu, který neoptimalizuje pozice pixelů (vlevo) a grafu, který pozice optimalizuje (vpravo).

5.3.3 Opakované kreslení cest, transformace, path clipping

Využívání cest má několik výhod. Pokud vykreslujeme výstup, který se pravidelně opakuje (například pravidelnou mřížku grafu), stačí vytvořit cestu pro jednu část mřížky. Tuto cestu uložit a poté ji vykreslit několikrát po sobě, jen s posunutím souřadnic vykreslení. Díky tomuto přístupu bude vykreslování mnohem rychlejší. CoreGraphics používá pokročilé metody kešování těchto uložených cest, takže výsledná rychlost kreslení je optimálnější, než kdybychom pokaždé cestu vytvářeli znova.

5.3.3.1 Posunutí a transformace grafického výstupu Posunutí souřadnic kreslení se děje pomocí funkce `CGContextTranslateCTM`, která ve svých parametrech očekává x a y souřadnici posunutí. Před každým voláním `CGContextTranslateCTM` by měl být uložen současný stav kontextu pomocí funkce `CGContextSaveGState`, která zároveň inicializuje nový grafický stav a dá ho na vrchol zásobníku grafických stavů. Poté, co je

vykreslování ukončeno, zavoláme funkci `CGContextRestoreGState`, změníme souřadnice posunutí na jinou pozici a celé vykreslení můžeme opakovat znova. Takto lze velmi rychle vykreslit opakující se grafické obrazce.

Posunutí je však jen jednou z transformací, které CoreGraphics nabízí. Vykreslený obrázek je možné například pootočit o určitý počet radiánů pomocí funkce `CGContextRotateCTM`, změnit poměry stran obrázku funkcí `CGContextScaleCTM` nebo na něj aplikovat jakoukoliv afinní transformaci. Afinní transformace se vytváří funkcí `CGAffineTransformMake`, které musíme předat hodnoty transformační matice $3x3$.

5.3.3.2 Omezení výstupu připínací cestou Dalším častým využitím cest je path clipping, tedy *připnutí* grafického výstupu dovnitř uzavřené cesty. Tohoto principu můžeme využít například pro vykreslení obdélníku se zaoblenými rohy. Stačí ke kontextu připnout cestu, která bude mít zaoblené rohy a pak kontext vykreslit. Zobrazí se jen oblast ohraničená cestou. Stejného přístupu se využívá i v grafové knihovně při kreslení grafu s výplní, který má barevně vyznačenu oblast pod datovou křivkou až po osu x.

Ukázková implementace path clipping ve sloupcovém grafu s výplní:

```
// 1. vytvoreni cesty
UIBezierPath *path = [UIBezierPath bezierPath];
[path moveToPoint:CGPointMake(0, CGRectGetHeight(rect))];
[path appendPath:[self closingPathForPlotWithIndex:i]];
[path addLineToPoint:CGPointMake(CGRectGetWidth(rect), CGRectGetHeight(rect))];
[path addLineToPoint:CGPointMake(0, CGRectGetHeight(rect))];
[path closePath];

// 2. ziskani vyplne
id<AreaFill> areaFill = [self.dataSource areaChart:self fillForPlotAtIndex:i];

// 3. ulozeni stavu, nastaveni clipping path, vykresleni
CGContextSaveGState(context);
[path addClip];
[areaFill drawInRect:CGRectOffset(rect, -rect.origin.x, -rect.origin.y)];
CGContextRestoreGState(context);
```

Výpis 21: Nastavení připínací cesty ve sloupcovém grafu

V prvním kroku ukázky vytvoříme připínací cestu rozšířením základní cesty ohraničující hodnoty v grafu o okrajové body grafu tak, aby byla cesta uzavřená a zahrnovala oblast od hodnot grafu po osu x. V druhém kroku zjistíme, jaká je výplň grafu, použitím metody Data Source příslušného grafu. Ve třetím kroku vytvoříme nový grafický stav, tomu připneme vytvořenou cestu a vykreslíme výplň zdánlivě přes celou oblast obdélníku, ve kterém kreslení probíhá. Díky nastavené připínací cestě se grafický výstup omezí jen touto cestou a nebude překrývat celý obdélník.

Ačkoliv jsou připínané cesty velkým pomocníkem při kreslení, jsou také častým zdrojem chyb. Mnohokrát se stává, že samotné kreslení je správné, ale programátor omylem nesprávně nastaví připínací cestu a výstup je pak úplně jiný, než jsou jeho představy. Pro odladění připínacích cest se doporučuje samotnou cestu vykreslit, abychom

zjistili, na jakou oblast jsme výstup omezili. Vykreslení cesty se docílí jednoduše funkcí `CGContextStrokePath`.

6 Přehled podporovaných grafů v knihovně

Tato část textu slouží jako rekapitulace, otestování a shrnutí prací, které jsem na grafovém frameworku vykonával. Zmíním zde kompletní výčet všech grafů a jejich ukázky. Pro jejich otestování jsem vytvořil aplikaci zobrazující náhodná data.

6.1 Koláčový graf

Koláčový graf obsahuje bez pochyby nejvíce propracované animace a reakce na dotyková gesta ze všech grafů obsažených v knihovně. Strávil jsem opravdu velké množství času optimalizováním animací tak, aby se pokud možno co nejvíce podobaly reálnému světu. Proto jsou v grafu zakomponovány nejrůznější fyzikální principy. Dokonce si troufám tvrdit, že animace tohoto grafu jsou kvalitnější, než animace koláčového grafu obsaženého v aplikaci Roambi, která mi byla pro tvorbu tohoto grafu vzorem.

6.1.1 Animace a efekty koláčového grafu

Graf obsahuje čtyři různé efekty:

- Při dotknutí se koláče a tažení prstem se koláč otáčí a podává informace o změně aktuálně vybrané výseče, na kterou zobáček grafu ukazuje.
- Při přerušení tažení a pozvednutí prstu z displeje graf pokračuje v rotaci zpomalujícím se pohybem a stále poskytuje informace, kterými výsečemi pohyb aktuálně prochází. V čím větší rychlosti byl pohyb tažení prstem ukončen, tím déle bude setrvačné otáčení probíhat.
- Pokud se graf setrvačně otáčí tak pomalu, že už zobáček grafu nedojede do další výseče, změní se efekt otáčení na simulaci magnetického přitahování středu poslední výseče k zobáčku grafu. Graf kolem tohoto zobáčku několikrát zakmitá. Počet a rychlost kmitů je závislá na velikosti výseče a rychlosti, s jakou se koláč před započetím efektu pohyboval.
- Při setrvačném pohybu se koláč dá zabrzdit také dotknutím prstu kdekoliv na ploše koláče. V této fázi koláč jako by byl brzděn tímto dotekem. Začne zpomalovat buď to dokud se stisknutí prstu neuvolní, pak pokračuje v dokončení setrvačnosti aktuální rychlostí. Nebo pokud se zastaví úplně, čeká na uvolnění prstu, po kterém se animací magnetického přitahování posune do středu aktuální kruhové výseče.

6.2 Spojnicový graf

Toto je další z notoricky známých a nejčastěji používaných typů grafů. Zobrazí se tak, že hodnoty vynesené mezi souřadnicemi x a y se jednoduše spojí rovnou čarou. Výsledná čára se tedy klikatí podle zobrazených dat. V dalších verzích tohoto grafu by mohla být přidána podpora interpolovaného zaoblení rohů křivky, které jsou nyní ostré. Graf by vypadal na pohled úhledněji. Zatím však zadavatel tento požadavek nevznesl.

Pomocí tohoto grafu je možné zobrazit více datových křivek. V takovém případě se rozmezí hodnot na osách grafu automaticky upraví tak, aby se do něj všechny křivky vešly. Osy x a y zobrazují hodnoty pouze některých dílků, vybírané automaticky podle dostupných dat. Zobrazují se například pouze násobky 10 nebo jiná vhodná čísla, aby byl graf dobře čitelný. Pokud hodnoty přesahují nulu, je nula ukázána vždy. Volitelně může graf v nulové hodnotě vykreslit vodorovnou čáru napříč celým prostorem.

6.2.1 Animace a efekty spojnicového grafu

Při animovaném zobrazení spojnicového grafu jeho křivky začínají s nulovou délkou a postupně se prodlužují, jako by je někdo kreslil tažením štětce. Tažení je započato větší rychlostí, v závěru se rychlost exponenciálně zpomaluje.

Graf také reaguje na dotyková gesta. Dotykem prstu se pomocí svislé čáry zvýrazní hodnota na ose x a barevnými body se vyznačí hodnoty na křivkách korespondující s x pozicí.

6.3 Spojnicový graf s výplní

Tento typ grafu je pouze odvozením z předešlého spojnicového grafu. Vznikne tak, že se oblast pod křivkami k ose x vybarví barevnou výplní. Kromě jednolité barvy graf podporuje výplň gradientem složeným ze dvou barev a v budoucnu je možnost přidání výplně vytvořené z obrázku. Výplni je možno nastavit také okraj libovolné tloušťky a barvy.

6.3.1 Animace a efekty spojnicového grafu s výplní

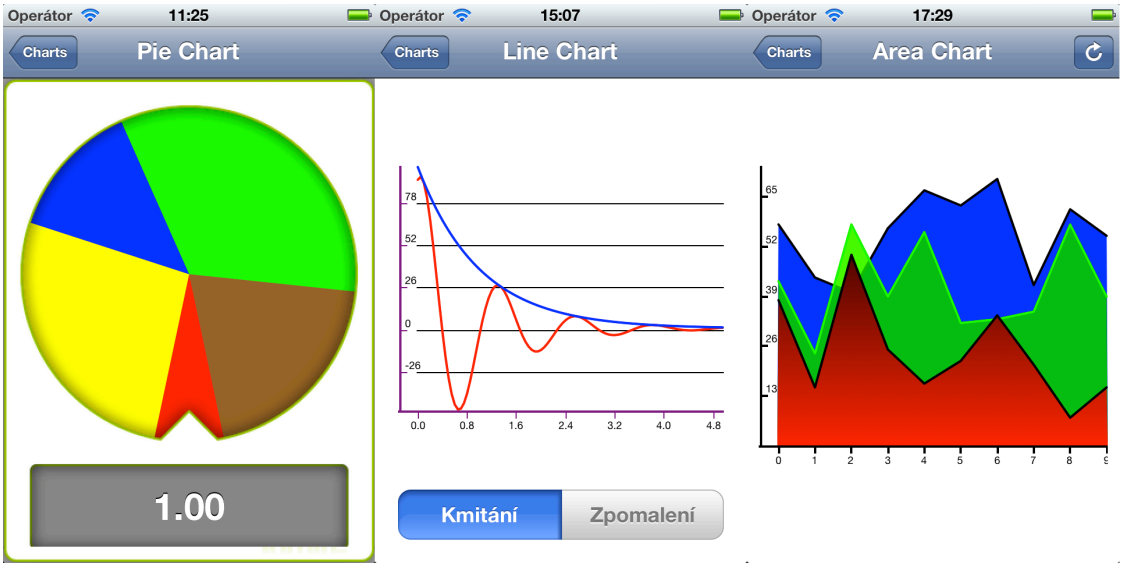
Pro tento typ grafu jsem zvolil jiný efekt animovaného zobrazení, než u jednoduchého spojnicového grafu. Animace opět probíhá zleva doprava, na začátku jsou všechny hodnoty jakoby zmáčknuty na malém prostoru a tento prostor se postupně rozšiřuje. Animace tak tvoří efekt, jako by se otevíral papír poskládaný do varhánek.

6.4 Sloupcový graf

Poslední z trojce nejžádanějších grafů v knihovně. Zobrazuje sloupce, jejichž výška se automaticky upraví podle dat grafu. Šířka sloupců se spočítá dynamicky podle počtu hodnot na ose x tak, aby sloupce měly vždy určitý poměr mezi tloušťkou sloupce a prázdným místem mezi dvěma sloupci. Tento poměr je 5:7, 5 dílků je široký sloupec, 2 dílky je mezera. Tyto rozměry jsou navíc ještě upraveny tak, aby zapadly přesně do pozic pixelů a všechny hrany tak zůstaly ostré. Výplň sloupců je opět možné nastavit buď to jednobarevnou nebo jako barevný přechod s volitelným okrajem.

6.4.1 Animace a efekty sloupcového grafu

Povaha sloupcového grafu se přímo vybízí k tomu, aby animace připomínala nárůst sloupců z nuly do své skutečné výšky. Proto jsem takovou animaci zvolil. Všechny sloupce



Obrázek 7: Ukázka grafů koláčového, spojnicového a spojnicového s výplní

najíždějí současně a všechny dokončí animaci ve stejný okamžik, nezávisle na své výšce. Není to tedy tak, že by se nižší sloupce zastavily dříve, než ty vyšší.

6.5 Histogram

Tento graf slouží jako základní pomůcka statistických analýz. Existují různé způsoby, jak zobrazovat histogram. V grafové knihovně je tento graf implementován jako spojení sloupcového grafu zobrazujícího hodnoty statistické proměnné, doplněného o křivku spojnicového grafu, na které je zaznamenána kumulativní četnost hodnot proměnné. Každý bod křivky četnosti má rovněž svůj popisek vyjadřující jeho procentuální hodnotu. Tato křivka je však dobrovolná a nemusí se zobrazovat, pokud ji programátor explicitně vypne použitím proměnné grafu.

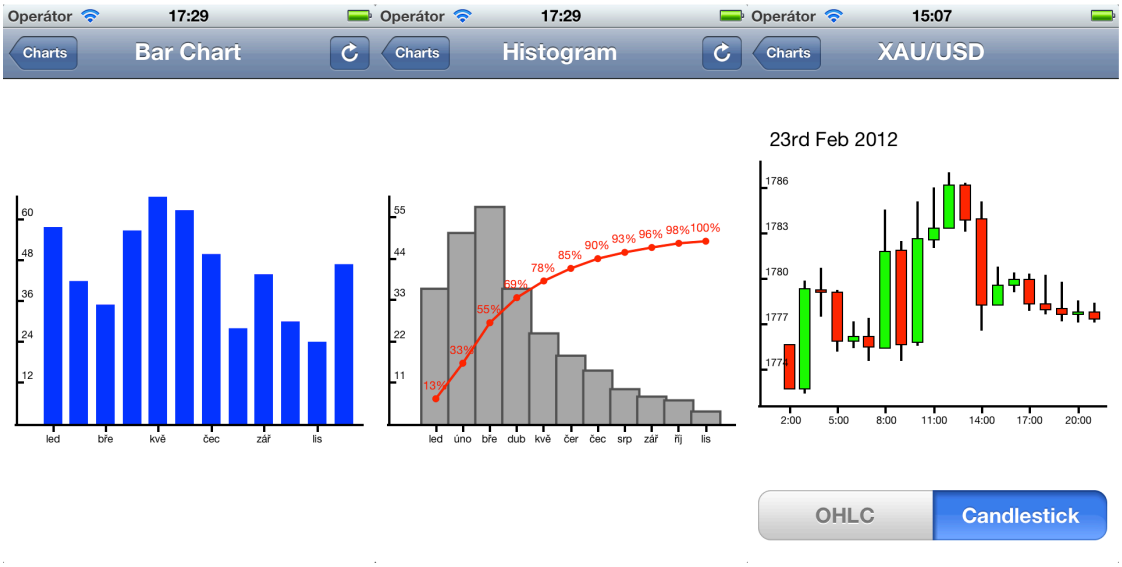
Graf umožňuje nastavit výplň sloupců a barvu kumulativní křivky. Sloupce grafu jsou zobrazeny vedle sebe, bez mezer.

6.5.1 Animace a efekty histogramu

Animace při načítání histogramu je složená ze tří částí. První dvě probíhají současně. Je to zvětšování sloupců a zároveň postupné kreslení křivky kumulativní četnosti. V závěru se zobrazí také samotná čísla popisků četností a tím se animace ukončí.

6.6 OHLC svíčkový burzovní graf

OHLC graf a svíčkový graf jsou ve skutečnosti dva různé typy, které však zobrazují stejná data – jde vždy o čtyři hodnoty pro danou x souřadnici – otevírací, nejvyšší, nejnižší



Obrázek 8: Ukázka sloupcového grafu, histogramu a OHLC svíčkového grafu

a uzavírací. Z tohoto důvodu jsem se rozhodl tyto dva grafy spojit do jednoho a pomocí booleovské proměnné lze grafu přikázat, jestli se má zobrazit jako OHLC nebo jako svíčkový. Způsob zobrazení těchto dvou grafů je popsán v kapitole o požadavcích na grafovou knihovnu.

6.6.1 Animace a efekty OHLC svíčkového grafu

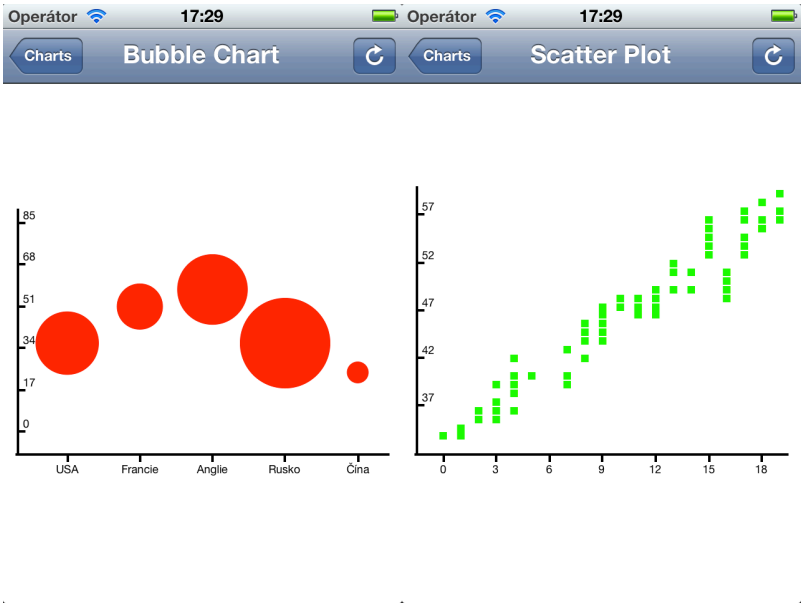
Graf se načítá s efektem, jako by jeho hodnoty vyrostly ze svého středu, rovnoměrně nahoru a dolů. Nereaguje na dotyková gesta.

6.7 Bublinový graf

Tento graf byl náročnější na implementaci z toho důvodu, že bubliny mohou mít proměnlivé rozměry. Nejprve bylo potřeba projít všechny hodnoty grafu, z těch spočítat minimální a maximální rozmezí y osy, a poté toto rozmezí zvětšit tak, aby dokázalo pojmout rozměry bublin, které by se do něj jinak nevešly. Rovněž rozmezí osy x se muselo upravit proto, aby bubliny nezasahovaly přes okraj grafu.

6.7.1 Animace a efekty bublinového grafu

Bubliny v grafu se zobrazují animovaně tak, že se z nuly zvětší do své skutečné podoby. Graf nereaguje na dotyková gesta.



Obrázek 9: Ukázka bublinového a korelačního grafu

6.8 Bodový korelační diagram

Anglickým názvem se tento diagram nazývá Scatter plot. Je velmi častým nástrojem statistických šetření. Na první pohled připomíná jednoduchý bodový diagram nebo spojnicový graf, jehož hodnoty ale nejsou spojeny. Rozdíl je však v tom, že korelační diagram může zobrazovat pro každou hodnotu na ose x libovolný počet bodů na ose y. Proto byl u implementace tohoto typu grafu potřeba trošku jiný přístup než u implementace spojnicového grafu. Výsledný graf zobrazuje body korelačního diagramu jako barevné čtverečky, kterým je možné nastavit libovolnou barvu.

6.8.1 Animace a efekty bodového korelačního diagramu

Při animaci tohoto grafu se body jednotlivě zobrazují podle svého pořadí zleva doprava. Stejně jako u jiných animací, animace je zpočátku rychlejší a v průběhu se zpomaluje. Graf nereaguje na dotyková gesta.

7 Závěr

Cílem mé diplomové práce bylo navrhnout vlastní grafovou knihovnu, která doplňuje knihovny již existující. Průběh návrhu, implementace a testování knihovny je popsán v dřívějších kapitolách tohoto textu. Zde zmíním, jaký je stav aktuálního řešení, jaké jsou možnosti dalšího vývoje knihovny a ve kterých aplikacích je knihovna již používána.

7.1 V jaké fázi je aktuální řešení

V současné chvíli je knihovna plně připravena k použití. Splňuje základní požadavky zadavatele a obsahuje osm různých typů grafů. Je implementována její nejzákladnější verze, tedy vybrání některého z podporovaných typů grafů, přiřazení mu potřebných dat a jeho zobrazení. Všechny grafy také podporují animace a knihovna je sestavena tak, aby pro programátora bylo co nejjednodušší rozšířit seznam grafů dle potřeby o své vlastní.

Firma Inmite už knihovnu aktivně používá v některých svých projektech (zmíněných níže).

7.2 Možnosti dalšího vývoje

V budoucnu plánuji přidání více grafů a doplnění některých animací a interakcí s grafy. Také je zde možnost vytvořit předpřipravené view controllery, které budou spojovat více grafů dohromady a vytvářet specifické analýzy nad daty.

Grafy jsou sestaveny s důrazem na jednoduchost, a proto zatím záměrně nepodporují zobrazování popisků os, legendy a dalších údajů. V budoucnu počítám s doplněním těchto informací, pravděpodobně jako oddělené komponenty grafů.

Chci také prozkoumat možnosti napojení zdroje dat grafů na webové API, protože zobrazování dat získávaných online bude zřejmě majoritním způsobem použití knihovny. Rád bych vytvořil nějakou obecnou strukturu tříd a rozhraní, které programátorovi co nejvíce usnadní napojení dat grafů na webové API.

7.3 Aplikace, ve kterých se už knihovna používá

Knihovna je připravena pro použití v další verzi aplikace At Media[7]. Bohužel update aplikace At Media ještě nebyl schválen mezi uživatele, takže knihovna dosud nebyla otestována v produkčním prostředí.

[7] http://itunes.apple.com/ke/app/atmedia-sledovanost-tv-stanic/id395915483?mt=8

8 Reference

[1] Christopher Allen, Shannon Appelcline. *Iphone in Action: Introduction to Web and SDK Development*, Manning Publications Co., 2009, ISBN 193398886X.

[2] Erica Sadun. *The iPhone Developer's Cookbook, Building Applications with the iPhone SDK*, Addison-Wesley, 2009, SBN-10: 0-321-55545-7.

[3] Jonathan Zdziarski. *iPhone SDK Application Development, 1st Edition*, O'Reilly Media, Inc., 2009, ISBN-13: 978-0-596-15405-9.

[4] Apple Developer *iPhone OS Reference Library,* `http://developer.apple.com/iphone/library/` Apple Inc., 2010.

[5] Alan Cannistraro, Josh Shaffeer. *iPhone Applicaiton Development (Winter 2010)* Stanford University on iTunes U, 2010.

[6] How to Make iPhone Apps. *iPhone OS Design Patterns,* `http://howtomakeiphoneapps.com/2009/06/iphone-os-design-patterns/`

[7] Cocoa Dev Central. *Core Data Class Overview,* `http://cocoadevcentral.com/articles/000086.php`

[8] Apple Developer *Quartz 2D Programming Guide,* `http://developer.apple.com/library/mac/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/Introduction/Introduction.html`

[9] Apple Developer *OpenGL ES Programming Guide for iOS,* `https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESontheiPhone/OpenGLESontheiPhone.html`

A Ukázky zdrojových kódů

A.1 Vytvoření jednoduchého grafu pomocí frameworku CorePlot

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        _chartData = [NSArray arrayWithObjects:[NSNumber numberWithInt:10], [NSNumber
            numberWithInt:10], [NSNumber numberWithInt:20],
                [NSNumber numberWithInt:30], [NSNumber numberWithInt:30], nil];
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    CPTGraphHostingView *hostingView = [[CPTGraphHostingView alloc] initWithFrame:self.view.
        bounds];
    CPTGraph *graph = [[CPTGraph alloc] init];

    CPTPieChart *pieChart = [[CPTPieChart alloc] init];
    pieChart.pieRadius = 70.0f;
    pieChart.centerAnchor = self.view.center;
    pieChart.dataSource = self;

    [graph addPlot:pieChart];

    [self.view addSubview:hostingView];
}

-(NSUInteger)numberOfRecordsForPlot:(CPTPlot *)plot {
    return [[self chartData] count];
}

-(NSNumber *)numberForPlot:(CPTPlot *)plot field:(NSUInteger)fieldEnum recordIndex:(
    NSUInteger)index {
    return (NSNumber *)[self.chartData objectAtIndex:index];
}
```

Výpis 22: Vytvoření jednoduchého grafu pomocí frameworku CorePlot

A.2 Vytvoření jednoduchého grafu pomocí frameworku PowerPlot

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    float sourceData[10] = { 0, 1, 7, 3, 4, 8, 6, 7, 4, 9 };
```

```
    data = [WSData dataWithValues:[WSData arrayWithFloat:sourceData
                                                 withLen:10]
                        withValuesX:[WSData arrayOfZerosWithLen:10]];
    data = [data indexedData];

    chartView = [[WSChart alloc] initWithFrame:CGRectMake(0, 0, CGRectGetWidth(self.view.
        bounds), CGRectGetHeight(self.view.bounds) / 2)];

    WSChart *plot = [[WSChart alloc] initWithFrame:CGRectInset(chartView.frame, 5, 20)];

    [plot configureWithData:data
                  withStyle:kChartLineScientific
              withAxisStyle:kCSPlain
            withColorScheme:kColorLight
                  withLabelX:@"X axis"
                  withLabelY:@"Y axys"];

    [chartView addSubview:plot];
    chartView.chartTitleFont = [UIFont systemFontOfSize:14];
    [chartView setChartTitle:@"Toto je muj graf"];

    [self.view addSubview:chartView];
}
```

Výpis 23: Vytvoření jednoduchého grafu pomocí frameworku PowerPlot

A.3 Vytvoření jednoduchého grafu pomocí knihovny iPhone Charting Library

```
- (void) drawRect: (CGRect) rect
{
    CGContextRef aContext = UIGraphicsGetCurrentContext();
    CGRect imageArea = CGRectMake(0, 0, 320, 400);

    IPCPieChart *pieChart = [[IPCPieChart alloc] initWith3D:NO];

    IPCTitle * title = [[IPCTitle alloc] initWithTitle :@"Muj kolacovy graf"];
    [pieChart setTitle : title ];

    DTCDefaultPieDataset *pDataset = [DTCDefaultPieDataset new];
    [pDataset setValueWithKey: (id <DTCIComparable>) @"Google" doubleValue: 84.96];
    [pDataset setValueWithKey: (id <DTCIComparable>) @"Yahoo" doubleValue: 6.24];
    [pDataset setValueWithKey: (id <DTCIComparable>) @"Bing/Live" doubleValue: 3.39];
    [pDataset setValueWithKey: (id <DTCIComparable>) @"Baidu" doubleValue: 3.06];
    [pDataset setValueWithKey: (id <DTCIComparable>) @"Others" doubleValue: 2.35];

    [pieChart drawChartWithContext:aContext area:imageArea dataset:pDataset];
}
```

Výpis 24: Vytvoření jednoduchého grafu pomocí knihovny iPhone Charting Library

A.4 Metoda vykreslující histogram pomocí animací

```
- (void)drawChartAreaInRect:(CGRect)rect {
    [super drawChartAreaInRect:rect];

    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextSaveGState(context);
    CGContextTranslateCTM(context, rect.origin.x, rect.origin.y);

    for (int plotIdx = 0; plotIdx < self.numberOfPlots; plotIdx++) {

        id<IMTAreaFill> areaFill = nil;
        if (_histoChartFlags.dsFillForBarAtIndexExits)
            areaFill = [self.dataSource histogram:self fillForBarAtIndex:plotIdx];

        if (areaFill == nil)
            areaFill = [IMTColorFill fillWithColor:[UIColor blackColor]];

        UIColor *color = nil;
        if (_histoChartFlags.dsColorForBarAtIndexhExists)
            color = [self.dataSource histogram:self colorForBarAtIndex:plotIdx];
        else if (areaFill.strokeColor != nil)
            color = areaFill.strokeColor;

        if (color == nil)
            color = [UIColor blackColor];

        CGFloat previousCumulativeXPosition = 0;
        CGFloat previousCumulativeYPosition = 0;

        for (int tickIdx = 0; tickIdx < self.numberOfXTicks; tickIdx++) {

            //draw bar
            CGPoint peak = [[[self.plotsYValues objectAtIndex:plotIdx] objectAtIndex:tickIdx]
                YValue].chartPosition;

            CGFloat barHeight = (CGRectGetHeight(rect) - peak.y) * _animationDistortion.height;
            CGRect barFrame = CGRectMake(peak.x - self.barsWidth / 2, -(barHeight -
                CGRectGetHeight(rect)), self.barsWidth, barHeight);

            CGContextSaveGState(context);

            [areaFill drawInRect:barFrame];

            UIBezierPath *path = [UIBezierPath bezierPath];
            [path moveToPoint:CGPointMake(CGRectGetMaxX(barFrame), CGRectGetMaxY(
                barFrame))];
            [path addLineToPoint:CGPointMake(CGRectGetMaxX(barFrame), CGRectGetMinY(
                barFrame))];
            [path addLineToPoint:CGPointMake(CGRectGetMinX(barFrame), CGRectGetMinY(
                barFrame))];
            [path addLineToPoint:CGPointMake(CGRectGetMinX(barFrame), CGRectGetMaxY(
                barFrame))];
```

```
if (areaFill.strokeWidth > 0 && areaFill.strokeColor != nil) {
    path.lineWidth = areaFill.strokeWidth;
    [areaFill.strokeColor setStroke];
    [path stroke];
}

CGContextRestoreGState(context);

//draw cumulative frequency
if (self.showCumulativeValue && _histoChartFlags.
    dsPercentageForCumulativeFrequencyAtChartPathExists) {

    NSIndexPath *chartPath = [NSIndexPath indexPathForAxisXTick:tickIdx inPlot:
        plotIdx];
    CGFloat maxHeight = CGRectGetHeight(rect) * 8.0/10.0;
    CGFloat y = CGRectGetHeight(rect);
    CGFloat cumulativePercentage = [self.dataSource histogram:self
        percentageForCumulativeFrequencyAtChartPath:chartPath];
    y -= maxHeight * cumulativePercentage;

    if (_histoChartFlags.dsColorForBarAtIndexhExists)
        color = [self.dataSource histogram:self colorForBarAtIndex:plotIdx];

    CGRect clippingRect = rect;
    clippingRect.size.width += self.barsWidth;
    clippingRect.size.width *= _animationDistortion.height;
    clippingRect.origin = CGPointMake(-self.barsWidth / 2, 0);

    CGContextSaveGState(context);
    CGContextAddRect(context, clippingRect);
    CGContextClip(context);

    CGContextSetLineWidth(context, 2.0f);

    if (tickIdx != 0) {

        CGContextMoveToPoint(context, previousCumulativeXPosition,
            previousCumulativeYPosition);
        CGContextAddLineToPoint(context, peak.x, y);
    }
    else {
        CGContextMoveToPoint(context, peak.x, y);
    }

    [color setStroke];
    CGContextStrokePath(context);

    //draw circle
    [[IMTDrawableArc arcWithCenter:CGPointMake(peak.x, y)
                            radius:3.0f
                        beginAngle:0.0f
                          endAngle:2 * M_PI
                         fillColor:color] drawInContext:context];
```

```
                //draw text
                if (_animationDistortion.height > 0.99f) {
                    NSString *label = [NSString stringWithFormat:@"%d%%", (int)(
                        cumulativePercentage * 100)];
                    UIFont *font = [UIFont systemFontOfSize:10];
                    CGSize labelSize = [label sizeWithFont:font];
                    CGRect labelFrame = CGRectMake(peak.x - labelSize.width / 2, y - labelSize.
                        height - 5.0, labelSize.width, labelSize.height);

                    [[IMTDrawableText textWithText:label withColor:color font:font inRect:
                        labelFrame] drawInContext:context];
                }

                CGContextRestoreGState(context);

                previousCumulativeXPosition = peak.x;
                previousCumulativeYPosition = y;
            }
        }
    }

    CGContextRestoreGState(context);
}
```

Výpis 25: Metoda vykreslující histogram pomocí animací

A.5 Kategorie UIView pro uložení pohledu do obrázku nebo PDF

Hlavičkový soubor: `UIView+UIViewExport.h`

```
#import "UIView+IMTUIViewExport.h"


@interface UIView (UIViewExport)

/** Returns representation of UIView as UIImage object */
- (UIImage *)imageRepresentation;

/** Exports view as PDf file stored at given path */
- (void)exportInPDFFileAtPath:(NSString*)path;

/** Exports view as PNG file stored at given path */
- (void)exportInPNGFileAtPath:(NSString*)path;

/** Exports view as JPEG file with compression quality stored at given path*/
- (void)exportInJPEGFileAtPath:(NSString*)path withQuality:(CGFloat)compressionQuality;

@end
```

Výpis 26: Kategorie UIView – hlavičkový soubor

Implementace: `UIView+UIViewExport.m`

```
#import "UIView+IMTUIViewExport.h"
#import <QuartzCore/QuartzCore.h>


@implementation UIView (IMTUIViewExport)

- (UIImage *)imageRepresentation {

    UIGraphicsBeginImageContextWithOptions(self.bounds.size, self.opaque, 0.0);
    [self.layer renderInContext:UIGraphicsGetCurrentContext()];

    UIImage* image = UIGraphicsGetImageFromCurrentImageContext();

    UIGraphicsEndImageContext();

    return image;
}

- (void)exportInPDFFileAtPath:(NSString*)path {

    CGRect mediaBox = self.bounds;
    CGContextRef ctx = CGPDFContextCreateWithURL((__bridge CFURLRef)[NSURL
        fileURLWithPath:path], &mediaBox, NULL);

    CGPDFContextBeginPage(ctx, NULL);
    CGContextScaleCTM(ctx, 1, -1);
    CGContextTranslateCTM(ctx, 0, -mediaBox.size.height);

    [self.layer renderInContext:ctx];

    CGPDFContextEndPage(ctx);
    CFRelease(ctx);
}

- (void)exportInPNGFileAtPath:(NSString*)path {

    NSData *imageData = UIImagePNGRepresentation([self imageRepresentation]);

    [imageData writeToFile:path atomically:YES];
}

- (void)exportInJPEGFileAtPath:(NSString*)path withQuality:(CGFloat)compressionQuality {

NSData *imageData = UIImageJPEGRepresentation([self imageRepresentation],
     compressionQuality);

    [imageData writeToFile:path atomically:YES];
}

@end
```

Výpis 27: Kategorie UIView – implementační soubor

B Užitečné odkazy

URL	Jakou informaci odkaz poskytuje?
http://developer.apple.com/iphone/	Oficiální dokumentace společnosti Apple, rozsáhlé množství příkladů
http://itunes.stanford.edu/	Kurzy programování iPhone aplikací od Stanfordské univerzity na iTunes U
http://cvut.iknow.eu/iphone/	České kurzy programování pro iPhone vyučované na ČVUT
http://iPhoneDeveloperTips.com/	Krátké tutoriály na nejrůznější témata z oblasti vývoje iPhone a iPad aplikací
http://icodeblog.com/	Další zdroj užitečných tutoriálů
http://iphonedevelopmentbits.com/	Blog shromažďující mnoho užitečných článků souvisejících s vývojem iPhone aplikací
http://www.cimgf.com/	Články na obtížnější témata týkající se vývoje iOS aplikací

C Obsah přiloženého CD

/src – složka se zdrojovými kódy

/src/lib – zdrojové kódy grafové knihovny

/src/samples – ukázkové aplikace používající knihovnu

/text – složka s tímto textem